CLOUD COMPUTING
Cloud Applications

Zeinab Zali
Isfahan University Of Technology

(References: Dan C. Marinescu - Cloud Computing_ Theory and Practice, http://storm.apache.org/, Coursera cloud computing course-Professor Indranil Gupta)

Data streaming concepts

Data streaming

- Data streaming is the transfer of data at a steady high-speed rate, with low and well-controlled latency
  - There is very high data volume
  - decisions have to be made in real-time
- This data needs to be processed sequentially and incrementally on a record-by-record basis or over sliding time windows
  - used for a wide variety of analytics including correlations, aggregations, filtering, and sampling

Data streaming Examples

- log files generated by customers using mobile or web applications
- information from social networks
  - e.g., Twitter real-time search
- Website statistics
  - e.g., Google Analytics
- Packet processing in Intrusion detection systems
  - Also processing alerts in IDS, e.g., in most datacenters

Stream vs batch processing

Stream processing	Batch processing
individual records or micro batches	large data batches
only the most recent data or data over a rolling time window	the entire, or a large segment of a data set
latency of milliseconds	Latency of minute or hours
simple response functions, aggregates, and rolling metrics	carrying out complex analytics
hard to reason about the global state	well-defined system state to checkpoint and later restart the computation

Well-known streaming tools(I)

- Apache Storm: It holds true streaming model for stream processing via core storm layer
  - can be created in Java, Scala, and Clojure
- Apache Spark: It acts as a wrapper over the batch processing
  - can be created in Java, Python, Scala, and R

Well-known streaming tools(II)

- MillWheel: a framework for building low-latency data-processing applications that is widely used at Google
  - Users specify a directed computation graph and application code for individual nodes
  - the system manages persistent state and the continuous flow of records

Why not MapReduce?

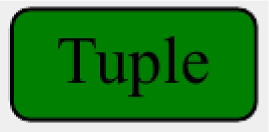
- MapReduce, Hadoop, etc., store and process data at scale, but not for real-time systems
- There's no hack that will turn Hadoop into a real-time streaming system
  - Fundamentally different set of requirements than batch processing

Storm

Storm Components

- Streams
- Spouts
- Bolts
- Stream groupings
- Topologies
- Reliability
- Tasks
- Workers

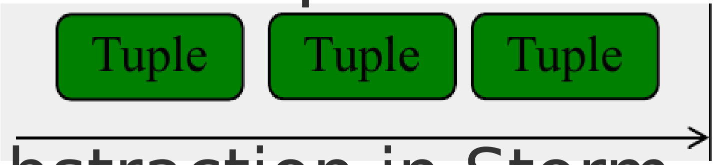
Tuple



- An ordered list of elements
- E.g., <tweeter, tweet>
  - E.g., <“Miley Cyrus”, “Hey! Here’s my new song!”>
  - E.g., <“Justin Bieber”, “Hey! Here’s MY new song!”>
- E.g., <URL, clicker-IP, date, time>
  - E.g., <coursera.org, 101.201.301.401, 4/4/2014, 10:35:40>
  - E.g., <coursera.org, 901.801.701.601, 4/4/2014, 10:35:42>

Stream

- A stream is an unbounded sequence of tuples that is processed and created in parallel in a distributed fashion



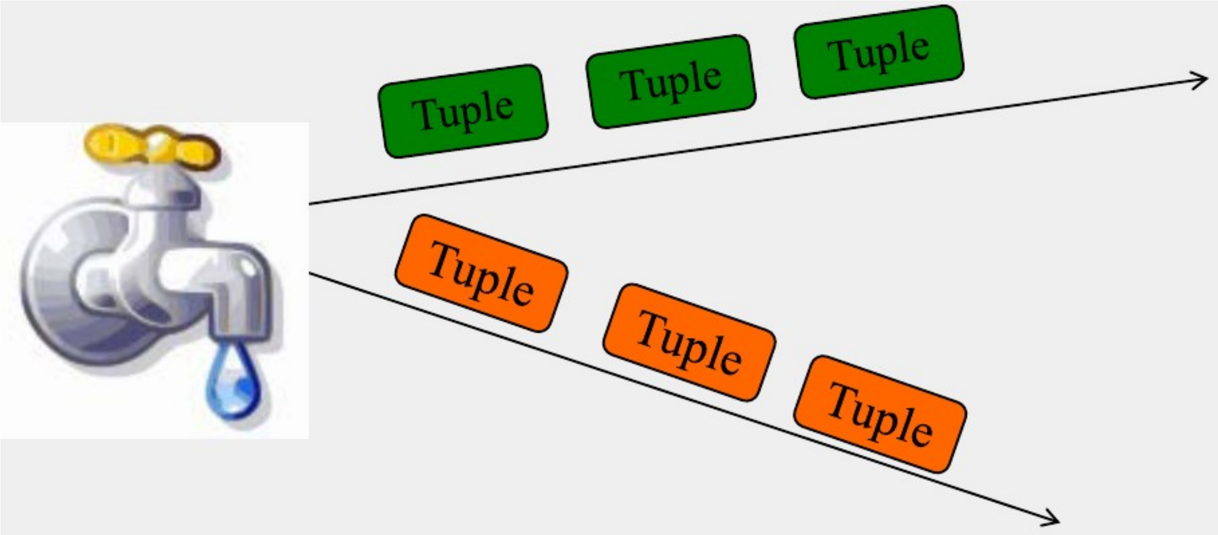
  - The stream is the core abstraction in Storm
- Social network example:
  - <“Miley Cyrus”, “Hey! Here’s my new song!”>,

    <“Rolling Stones”, “Hey! Here’s my old song that’s still a super-hit!”>, ...
- Website example:
  - <coursera.org, 101.201.301.401, 4/4/2014, 10:35:40>,

    <coursera.org, 901.801.701.601, 4/4/2014, 10:35:42>, ...

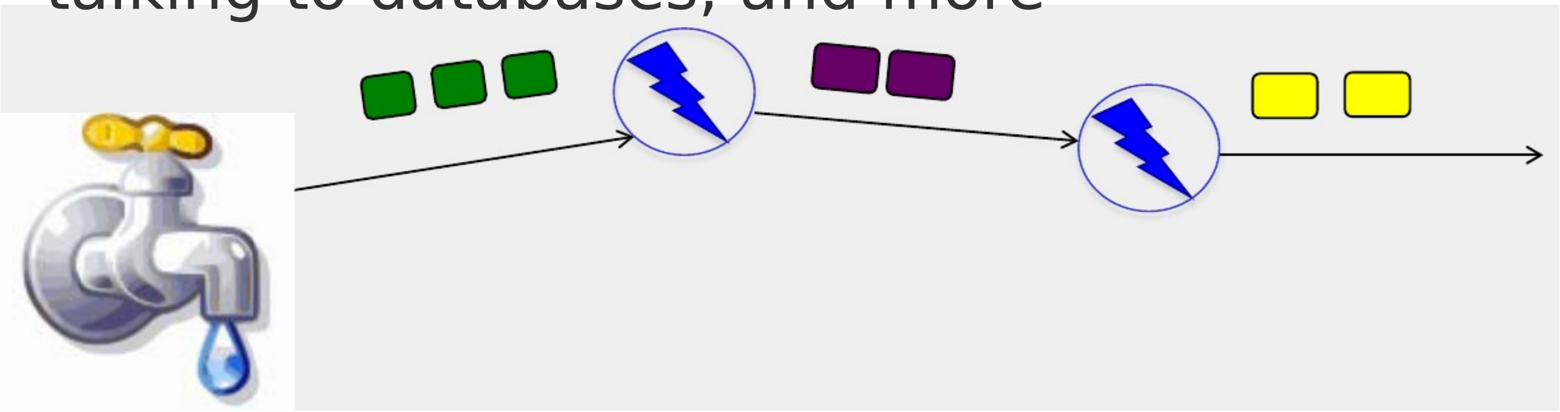
Spout

- A Storm entity (process) that is a source of streams
- Generally spouts will read tuples from an external source
  - Ex: from a crawler or DB



Bolt

- A Storm entity (process) that
  - Processes input streams
  - Outputs more streams for other bolts
- Bolts are the only entity in storm that can do processing, that is anything:
  - from filtering, functions, aggregations, joins, talking to databases, and more

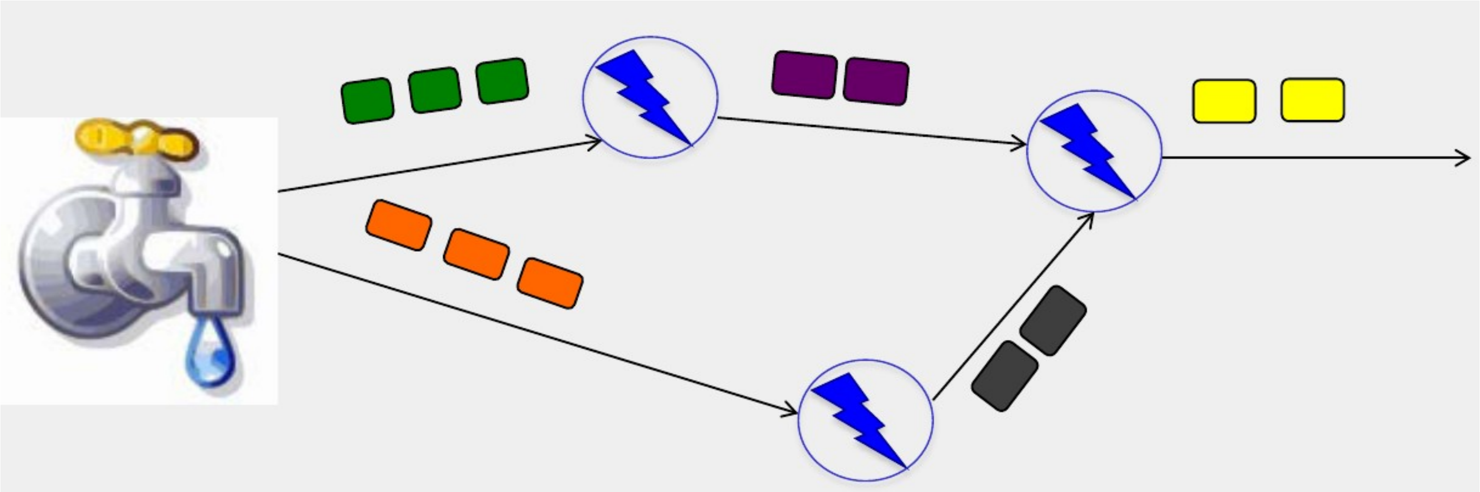


Bolts types

- Operations that can be performed
  - Filter: forward only tuples which satisfy a condition
  - Joins: When receiving two streams A and B, output all pairs (A,B) which satisfy a condition
  - Apply/transform: Modify each tuple according to a function
  - And many others
- bolts need to process a lot of data
  - Need to make them fast

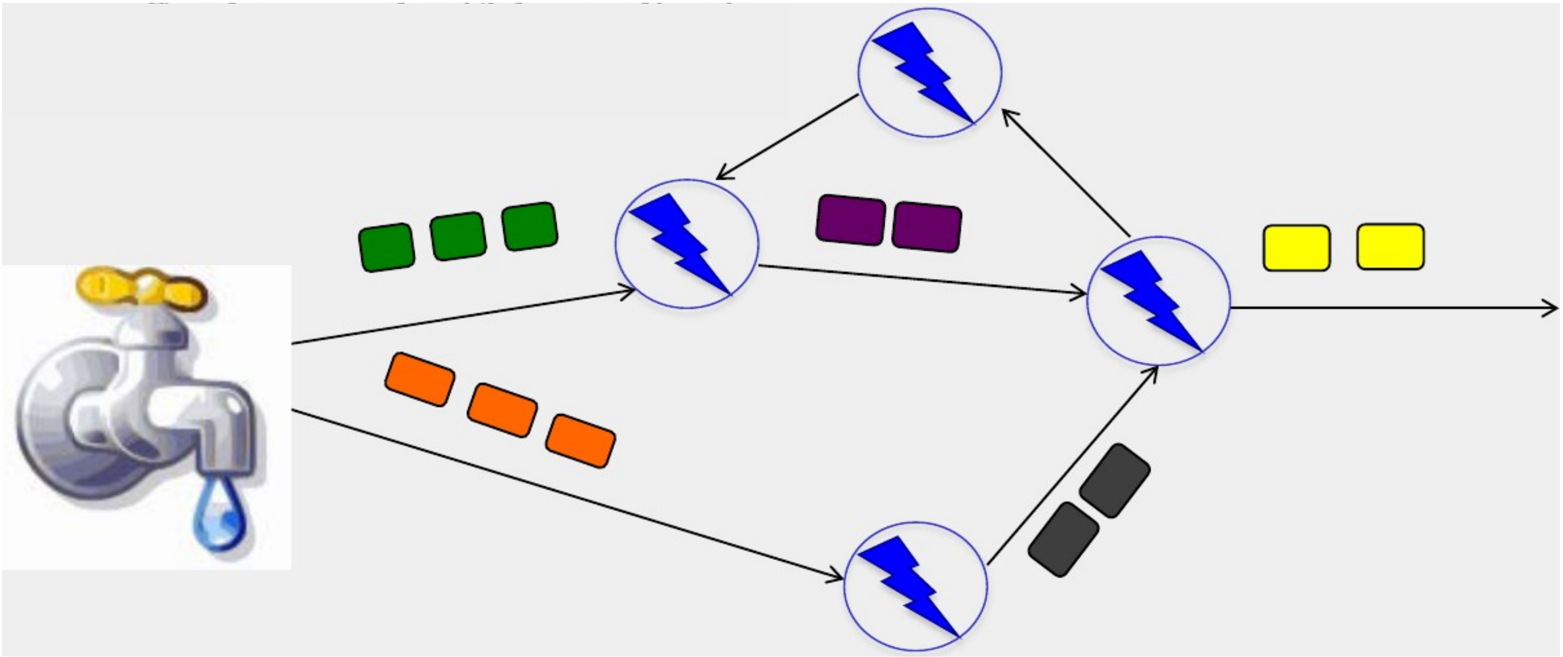
Topology (I)

- A topology is a graph of spouts and bolts
  - The logic for a realtime application is packaged into a Storm topology
  - A Storm topology is analogous to a MapReduce job
  - MapReduce job eventually finishes, whereas a topology runs forever (or until you kill it, of course)



Topology (II)

- Topology can have cycles if the application requires it



Parallelizing Bolts

- Have multiple processes (“tasks”) constitute a bolt
- Incoming streams split among the tasks
- Typically each incoming tuple goes to one task in the bolt
  - Decided by “Grouping strategy”

Stream Grouping

- Part of defining a topology is specifying for each bolt which streams it should receive as input
- A stream grouping defines how that stream should be partitioned among the bolt's tasks
- There are some built-in stream groupings in Storm, and you can implement a custom stream grouping

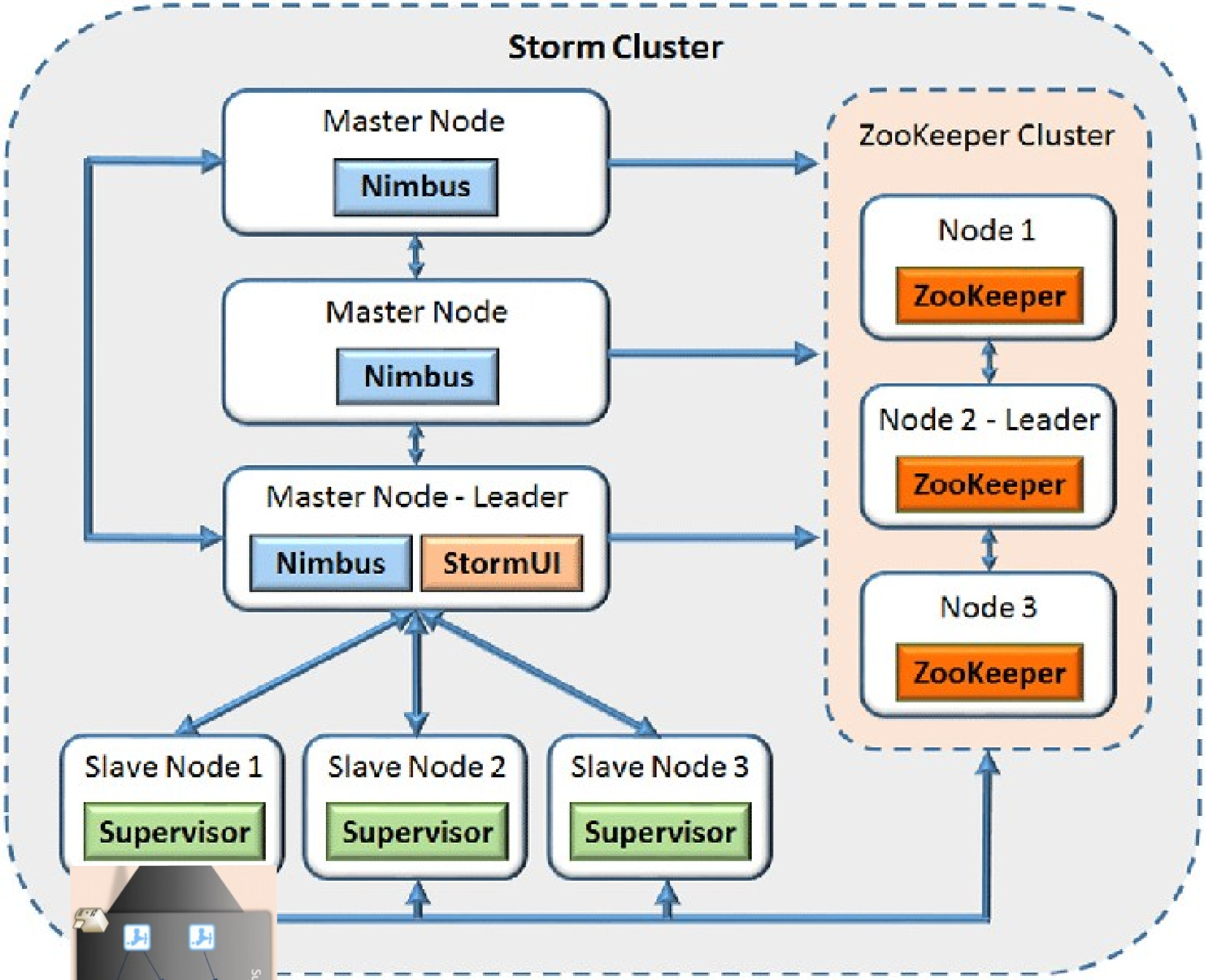
Stream Grouping Types

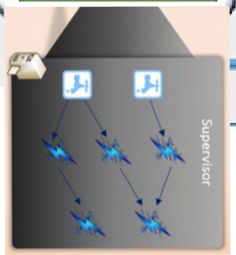
- Shuffle Grouping: Streams are distributed evenly across the bolt’s tasks
  - Ex: Round-robin fashion
- Fields Grouping: Group a stream by a subset of its fields
  - E.g., All tweets where twitter username starts with [A-M,a-m,0-4] goes to task 1, and all tweets starting with [N-Z,n-z,5-9] go to task 2
- All Grouping:
  - All tasks of bolt receive all input tuples
  - Useful for joins

Storm cluster

- Master node: Runs a daemon called Nimbus, Responsible for
  - Distributing code around cluster
  - Assigning tasks to machines
  - Monitoring for failures of machines
- Worker node: Runs on a machine (server)
  - Runs a daemon called Supervisor
  - Listens for work assigned to its machines
- Zookeeper: Coordinates Nimbus and Supervisors communication
  - All state of Supervisor and Nimbus is kept here

Storm architecture





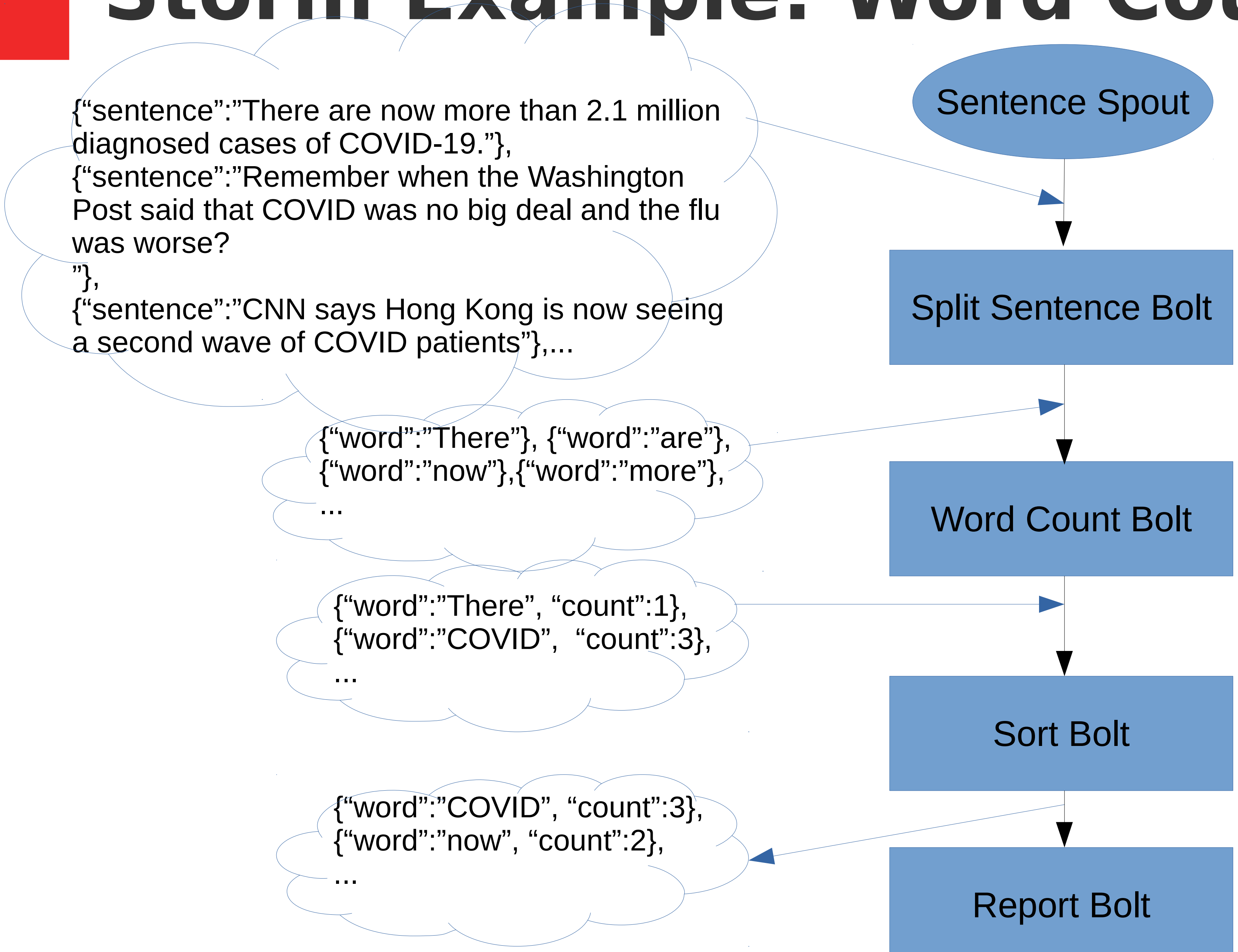
Failures and Reliability

- Spouts can either be reliable or unreliable
  - A reliable spout is capable of replaying a tuple if it failed to be processed by Storm
  - unreliable spout forgets about the tuple as soon as it is emitted
- A tuple is considered failed when its topology of resulting tuples fails to be fully processed within a specified timeout
  - Anchoring: Anchor an output to one or more input tuples, so Failure of one tuple causes one or more tuples to replayed

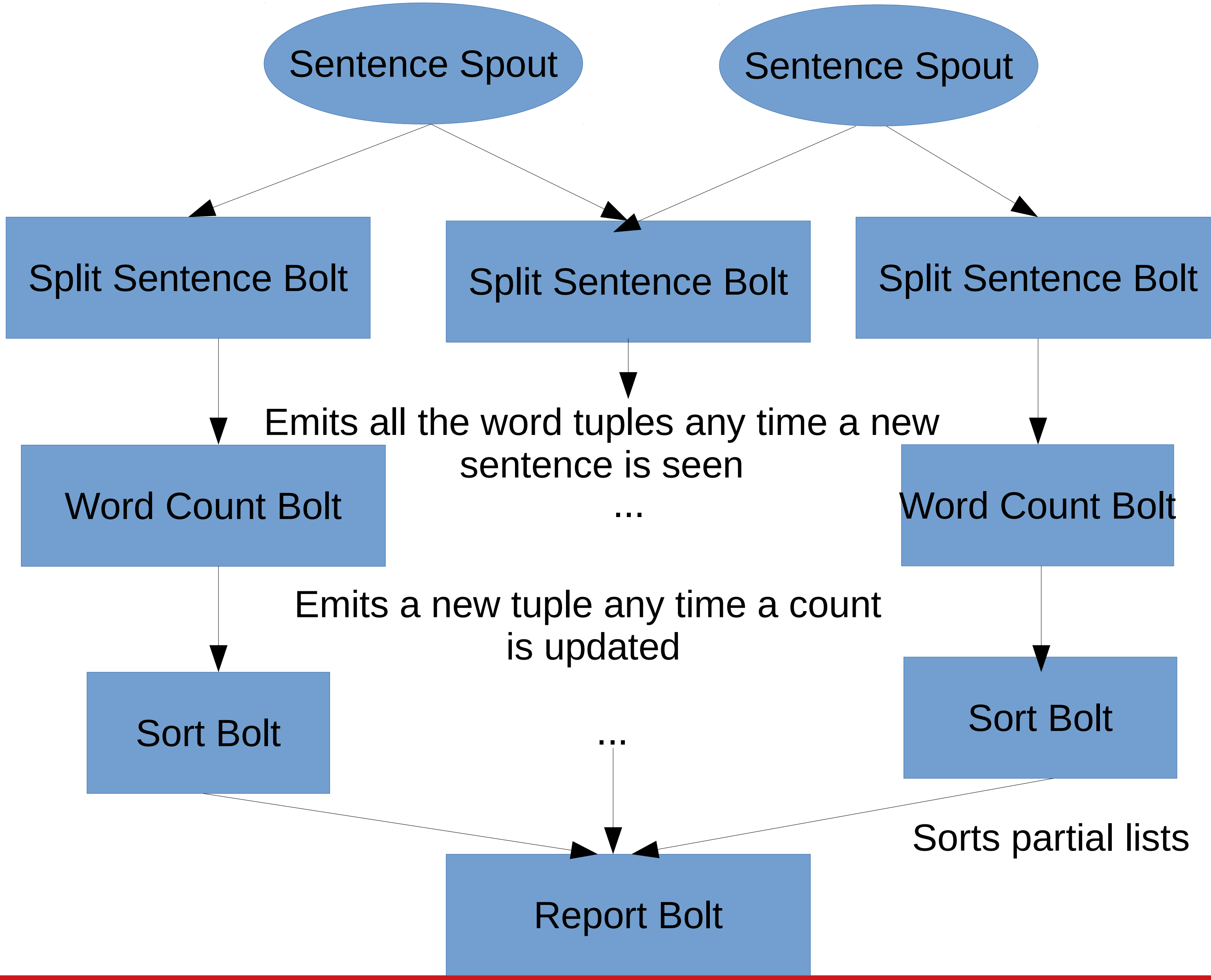
API For Fault-Tolerance

- Emit(tuple, output): Emits an output tuple, perhaps anchored on an input tuple
- Ack(tuple): Acknowledge that you finish processing a tuple
- Fail(tuple): Immediately fail the spout tuple at the root of tuple topology if there is an exception from the database, etc.
- Must remember to ack/fail each tuple
  - Each tuple consumes memory. Failure to do so results in memory leaks.

Storm Example: Word Count



Example: Parallelism in storm



Example: Programming

```
import twitter4j.*;
public class TwitterSampleSpout extends BaseRichSpout {
        private LinkedBlockingQueue<Status> queue;
        public TwitterSampleSpout(accessKeys) {
                .
                .
                .
        }

      @Override
      public void open(Map conf, TopologyContext context,
                              SpoutOutputCollector collector) {
            StatusListener listener = new StatusListener() {
                  .
                  .
                  .
            }
      }
      @Override
      public void nextTuple() {
              if ((status=queue.poll())!=NULL)
                  collector.emit(new Values(status));}
}
```

Example: Programming

```
public class SplitSentenceBolt extends BaseRichBolt{
        private OutputCollector collector;
        @Override
        public void prepare(Map config, TopologyContext
                                context, OutputCollector collector) {
            this.collector = collector;
        }
        @Override   //Code to split a sentence
        public void execute(Tuple tuple) {
           String sentence =      tuple.getStringByField("sentence");
           String[] words = sentence.split(" ");
           for(String word : words){
                 this.collector.emit(new Values(word));
           }
        }
         public void declareOutputFields(OutputFieldsDeclarer declarer) {
                    declarer.declare(new Fields("word"));}
}
```

Example: Programming

```
public class WordCountBolt extends BaseRichBolt {
    ...
    @Override
     public void prepare(Map config, TopologyContext
                                  context, OutputCollector collector) {
            this.collector = collector;
            this.counts = new HashMap<String, Long>();
     }
    @Override   //Code to count words
       public void execute(Tuple tuple) {
           String word = tuple.getStringByField("word");
           If ((count=this.counts.get(word))==null){
                count = 0;}
           count++;
           this.counts.put(word, count);
           this.collector.emit(new Values(word, count));
       }
      public void declareOutputFields(OutputFieldsDeclarer declarer) {
                    declarer.declare(new Fields("word", "count"));}  }
```

Example: Programming

```
public class SortBolt extends BaseRichBolt {
    ...
    private final List<WordCount> sortedWords = new
ArrayList<WordCount>(N);
    @Override
    public void prepare(){
        //new thread to call emition every t seconds
    }
    @Override   //Code to sort a collection
      public void execute(Tuple tuple) {
        String word = tuple.getStringByField("word");
        //search in a sorted collection (sortedWords)
        //and place it in its correct place
        ....

      }
     Private void emition(){
        this.collector.emit(sortedWords);}

}
```

```
public static class WordCount {
    String word; long count;
    public WordCount(String word,
                    long count) {
      this.word = word;
      this.count = count; }
}
```

Example: Programming

```
public class ReportBolt extends BaseRichBolt {
    ...
    private final List<WordCount> sortedWords = new
ArrayList<WordCount>(N);
    @Override   //Code to sort a collection
       public void execute(Tuple tuple) {
          PartialList sortedWords=tuple.getStringByField("list");
          //merge it with the global sortedWords

       }

      @Override   //Storm calls this method when a bolt is about to be shutdown
      public void cleanup() {

          //prints the sortedWords in a file or output

      }

}
```

Example: Programming

```
Public class wordCountTopology(){
    Public static void main(){
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout(1, new TwitterSampleSpout(accessKeys), 1);
        builder.setBolt(2, new SplitSentenceBolt(), 8).shuffleGrouping(1);
        builder.setBolt(3, new WordCountBolt(), 12).fieldGrouping(2, new
Fields("word"));
        builder.setBolt(4, new SortBolt(), 12).fieldGrouping(3, new
Fields("word"));
        builder.setBolt(5, new ReportBolt(), 1).globalGrouping(4);
        StormSubmitter.submitTopology("word count", builder.createTopology);
}
```

Spark streaming

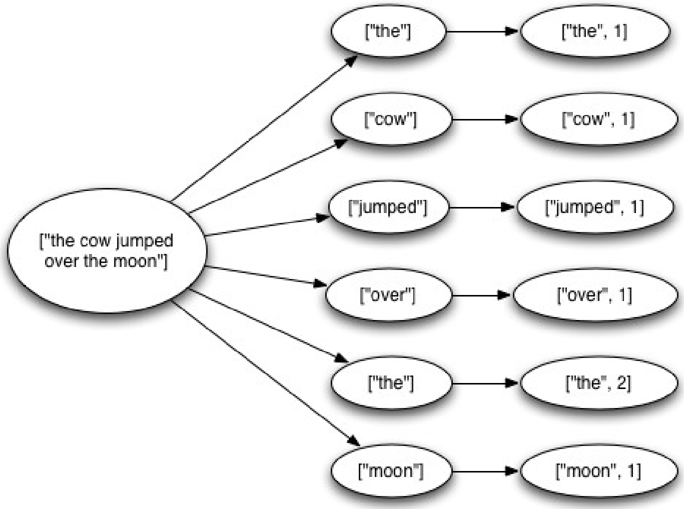
Resources:
Cloud Computing, Theory and practice, Dan.C. , chapter 12.6 Spark streaming
Spark documentation
Coursera, cloud computing applications, Reza Farivar

storm weakness (I)

- Traditional streaming systems like storm have a record-at-a-time processing model
  - Each node has mutable state
  - For each record, update state and send new records
  - State is lost if node dies!
- Anchoring in storm
  - Replay one or some anchored tuples if a tuple is failed to be processed

storm weakness (II)

Anchoring may result in
“not exactly once process”



- Storm Replays record if not processed by a node
  - May update mutable state twice!
  - Mutable state can be lost due to failure!

Spark vs storm (I)

- Spark streaming supports stateless operations acting independently in each time interval, as well as aggregation over time window
  - Window a bit of data
  - Run a batch
  - Repeat

Spark vs storm (III)

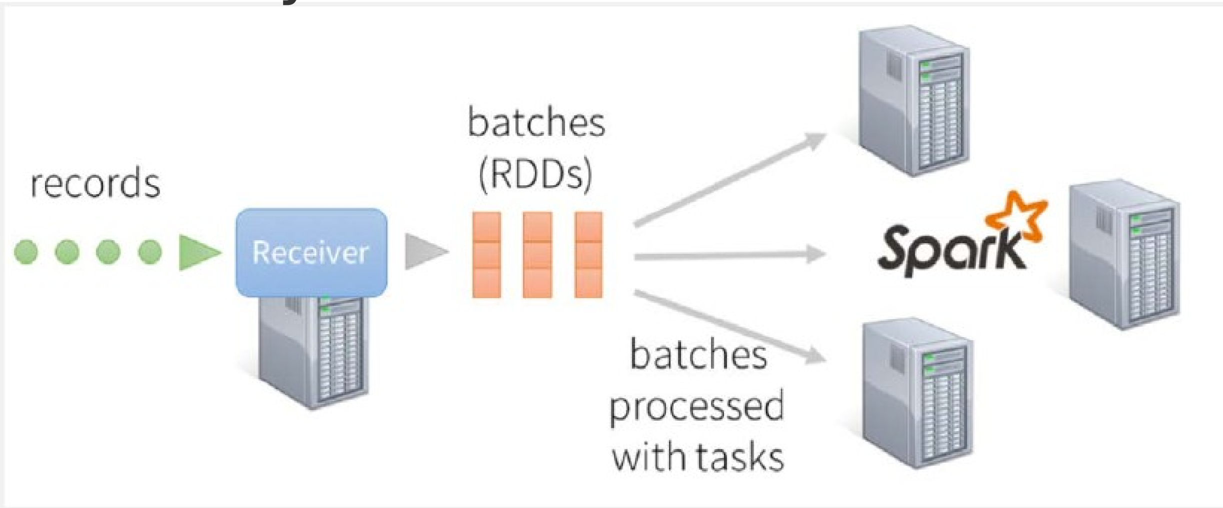
Features	Apache Storm	Apache Spark
Programming Language	Java, Scala, Clojure	Java, Scala,
Processing Models	True stream processing model through system layer.	Apache Spark Streaming is wrapper over batch processing.
Reliability	Supports "exactly once" processing mode. Can also be used in "at least once" and "at most once" processing modes.	Supports "exactly once" processing mode.
Latency	Apache Storm provides better latency	Apache Spark provides less latency
Resource Management	Can run on YARN and Mesos.	Can run on YARN and Mesos.

Spark project

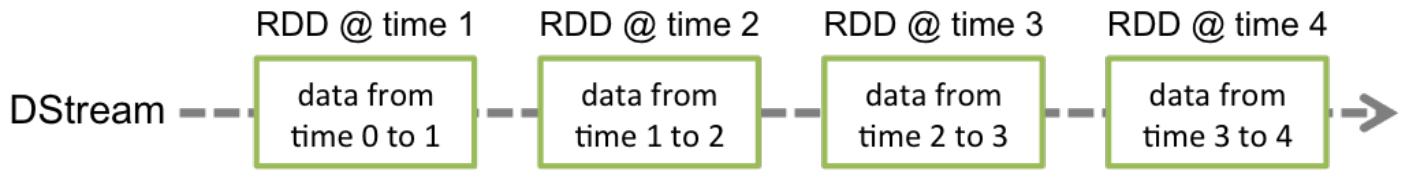
- Spark is Most contributed open source project in big-data domain (Berekely project)
  - It is a fast and general-purpose cluster computing system
  - It provides APIs in Java, Scala, Python and R
  - It supports a rich set of higher-level tools
    - Spark SQL for SQL and structured data processing
    - MLlib for machine learning
    - GraphX for graph processing
    - Spark Streaming

Resilient Distributed Dataset (RDD)

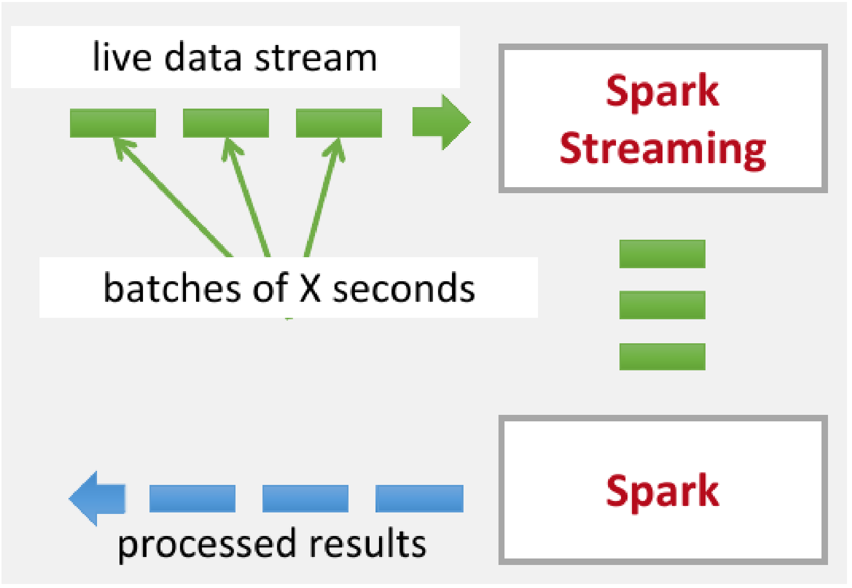
- The main abstraction Spark provides is RDD
- RDD is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel
  - RDD provides low latency
  - RDD provides ability to rebuild lost data without replication



Discretized stream (D-Stream)



- D-streams: streaming model in spark
  - Chops up the live stream into batches of X seconds
  - Spark treats each batch of data as RDDs and processes them using RDD operations
  - The processed results of the RDD operations are returned in batches
  - Batch sizes as low as 0.5s




```
public StreamingContext(SparkConf conf,
                        Duration batchDuration)
```

Spark usual operations

- transformations available in batch frameworks
  - Stateless and statefull transform operators
    - Stateless: they act independently on each time interval

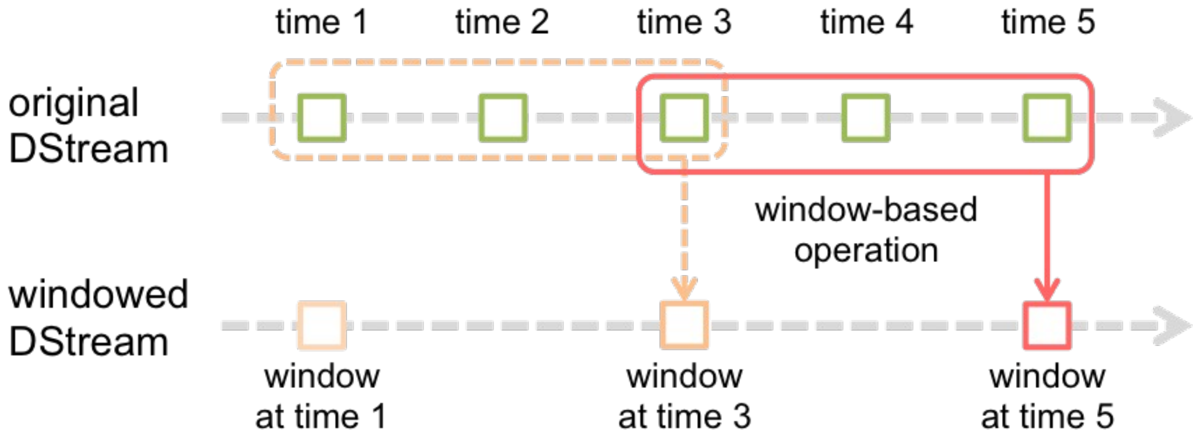
      map(func), join(otherStream), reduce(func), count(),...
    - Statefull: they share data among intervals

      updateStateByKey(func)
  - Output operators
    - they save data, e.g., store RDDs on HDFS

      saveAsHadoopFiles(prefix, [suffix])
      saveAsTextFiles(prefix, [suffix]),...

Spark streaming window operations

- Window: grouping records from a range of past intervals into one RDD.
  - allow to apply transformations over a sliding window of data
  - the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream



CountByWindow
(windowLength, slideInterval),
ReduceByWindow(
func, windowLength, slideInterval)
,...

DStream Input Sources

- Out of the box
  - Kafka
  - HDFS
  - Flume
  - Akka Actors
  - Raw TCP sockets
- Very easy to write a receiver for your own data source

Druid and Spark

- Druid is a column-oriented distributed data store that is ideal for powering user-facing data applications
  - Druid's focus is on extremely low latency queries
- Druid and Spark are complementary solutions as Druid can be used to accelerate OLAP queries in Spark
  - Druid fully indexes all data, and can act as a middle layer between Spark and your application