Introduction to large scale data mining: MapReduce, Spark, and Storm

Author: Dianwei Han

Center for Ultra-Scale Computing and
Information Security
EECS Department
Northwestern University

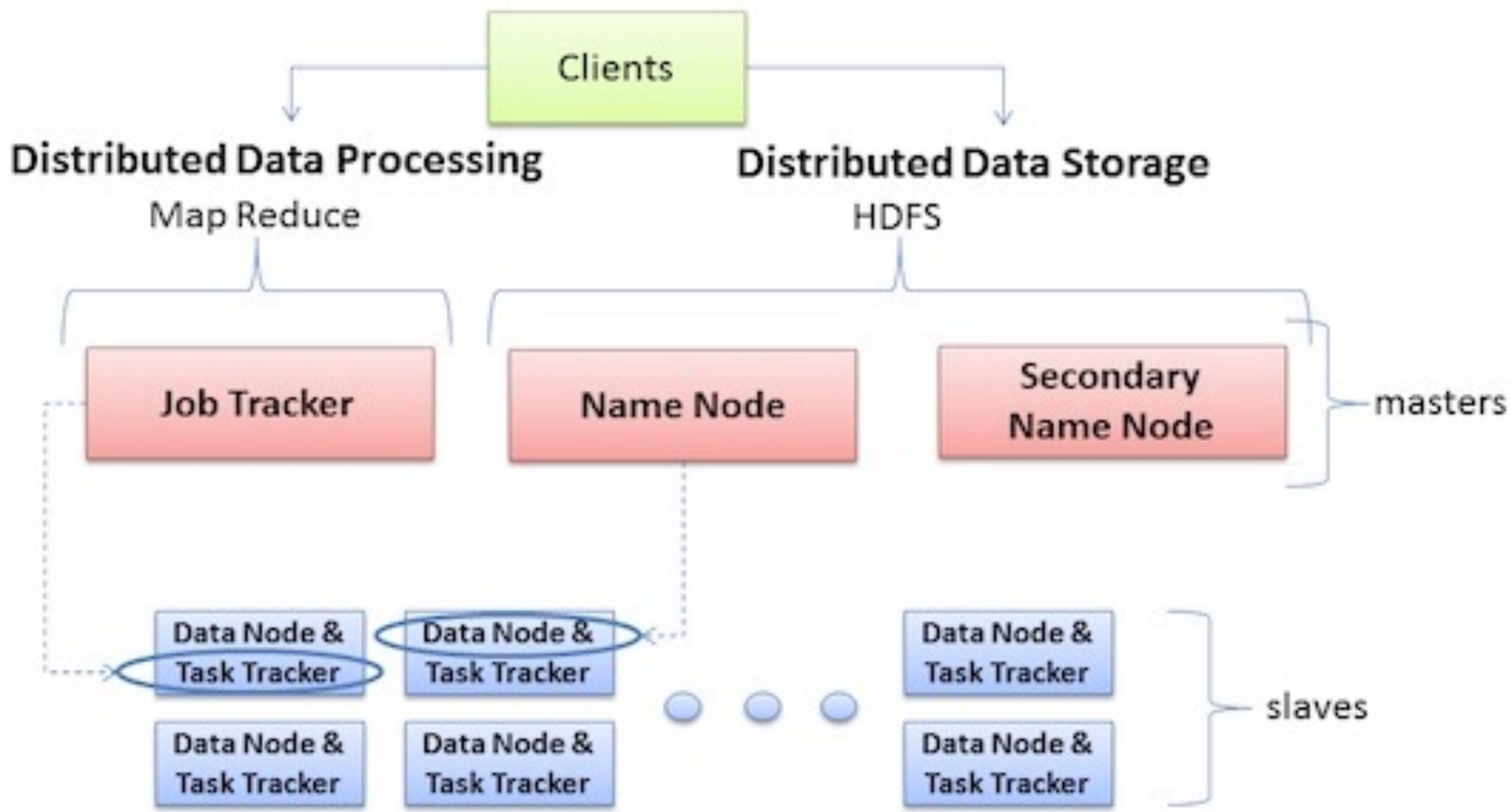
Outline:

- Big data framework
- MapReduce framework
- Spark Core framework
- MapReduce VS Spark core
- Storm framework
- Storm VS Spark Streaming

Big Data framework: Hadoop, Spark, Storm, and others

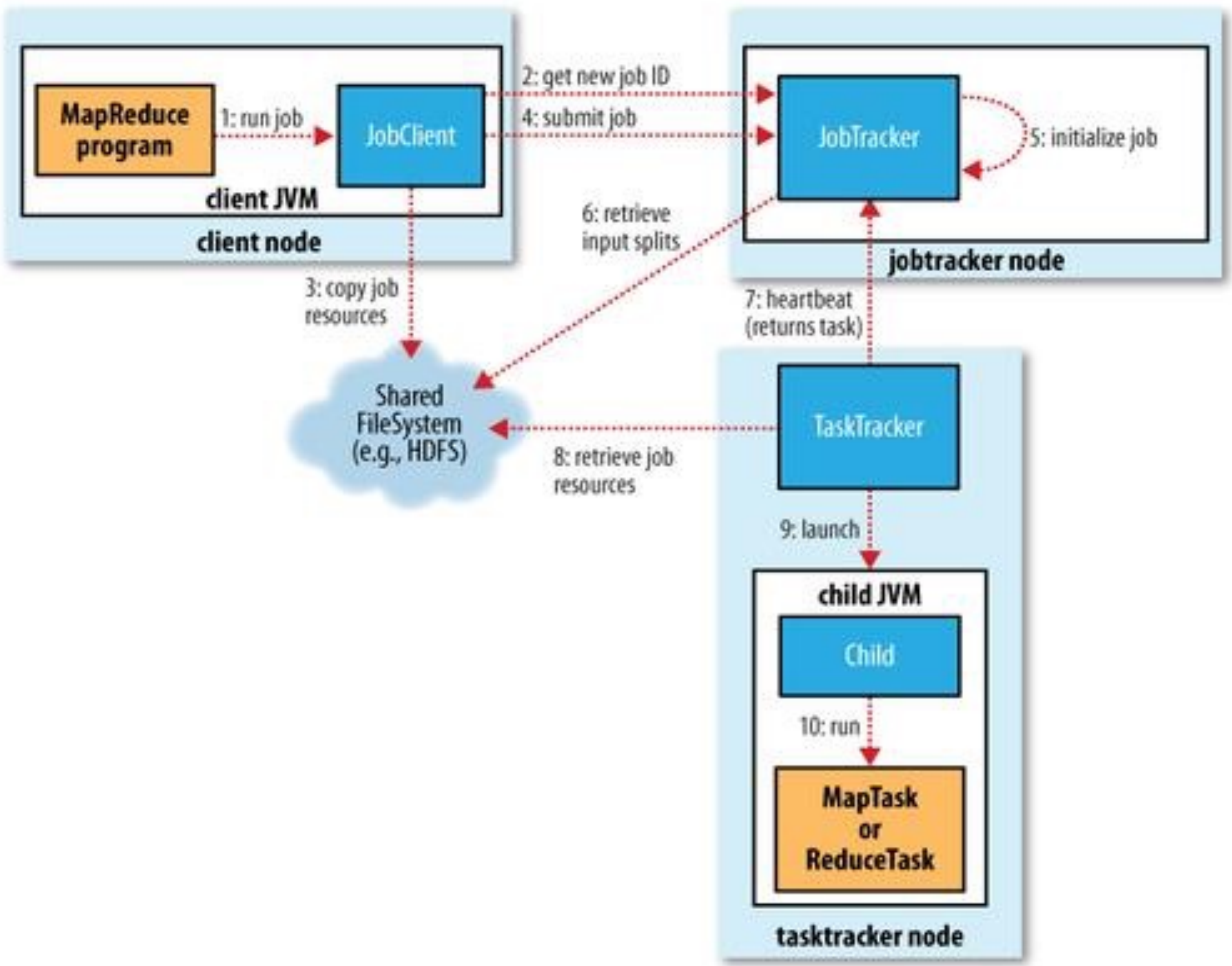
	Traditional RDBMS	Map Reduce
Data size	Gigabytes	PetaByes
Access	Interactive and batch	Batch and real time
updates	Read and write m times	Write once, read m times
structure	Static schema	Dynamic schema
scaling	nonlinear	linear

Master-Slave Architecture



Name node receives heartbeat from Data nodes. Job Tracker receives heartbeat from Task Trackers.

How Hadoop runs a Map/Reduce Job



Map/Reduce work flow

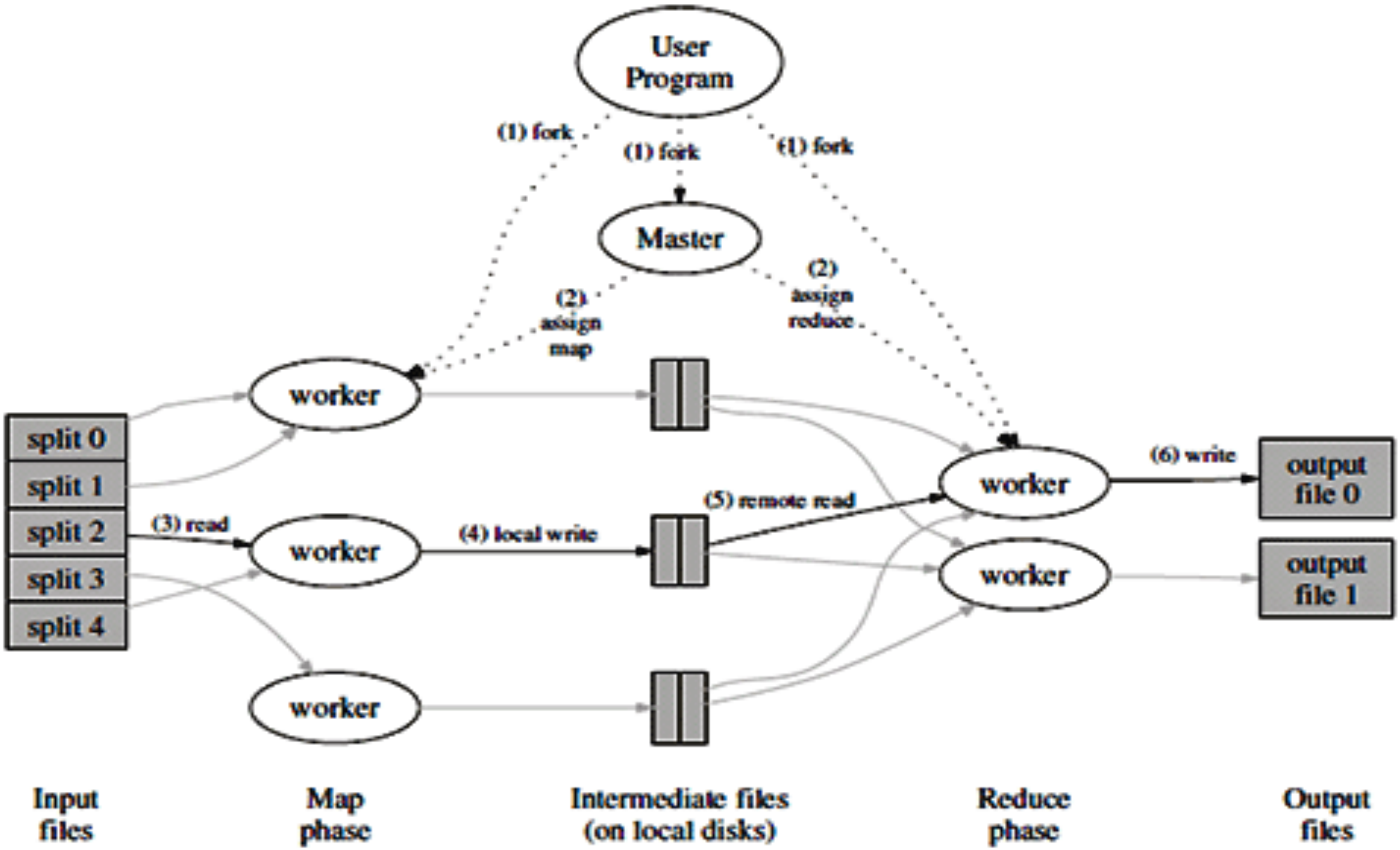
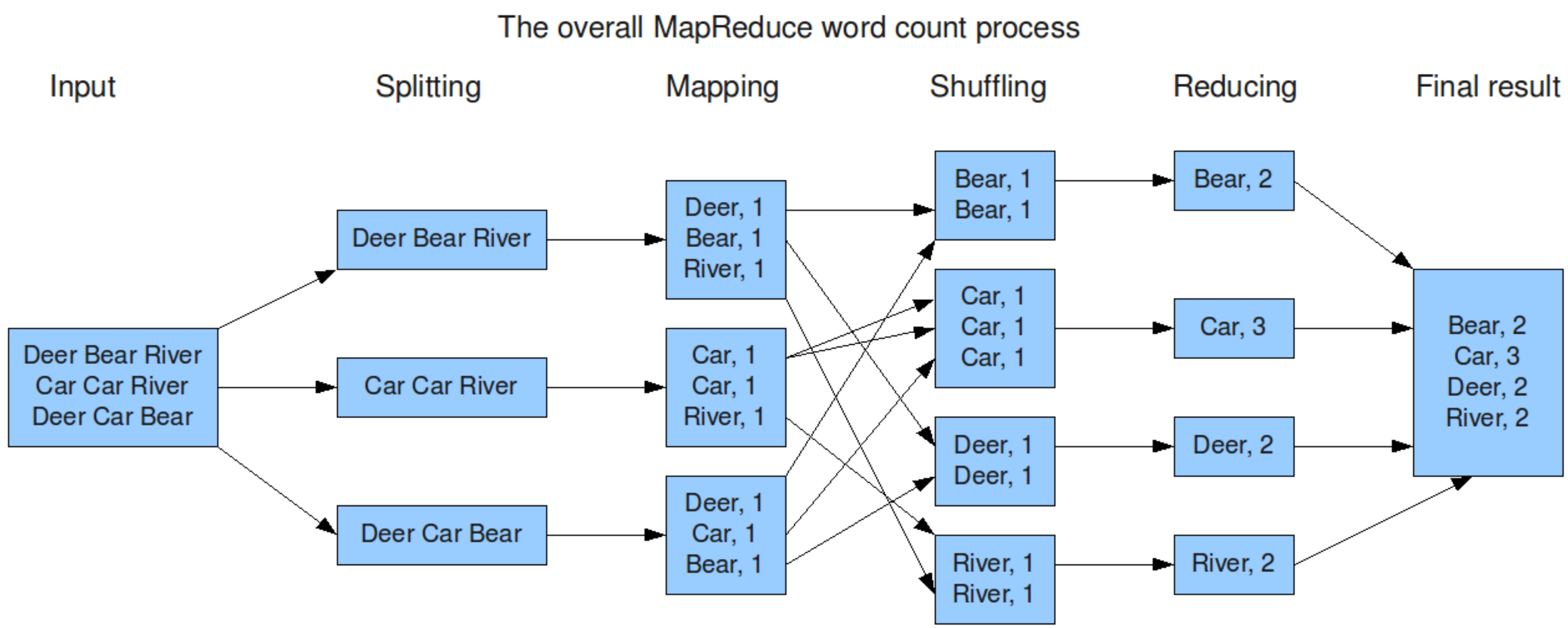


Figure 1: Execution overview

One example of using Map/Reduce



Features of hadoop

Scalable: easy to add new datanode.

Fault-tolerant: Failures are common, replications of block on datanode, task failed, job tracker find another task tracker to take the task.

Powerful: Big Data Computations that need the power of many computers, Large datasets: hundreds of TBs, tens of Pbs

Accessible: Open source

Reliable: multiple copies for block.

Master-Slave architecutre.

Features of hadoop (continue)

Perfect? NO

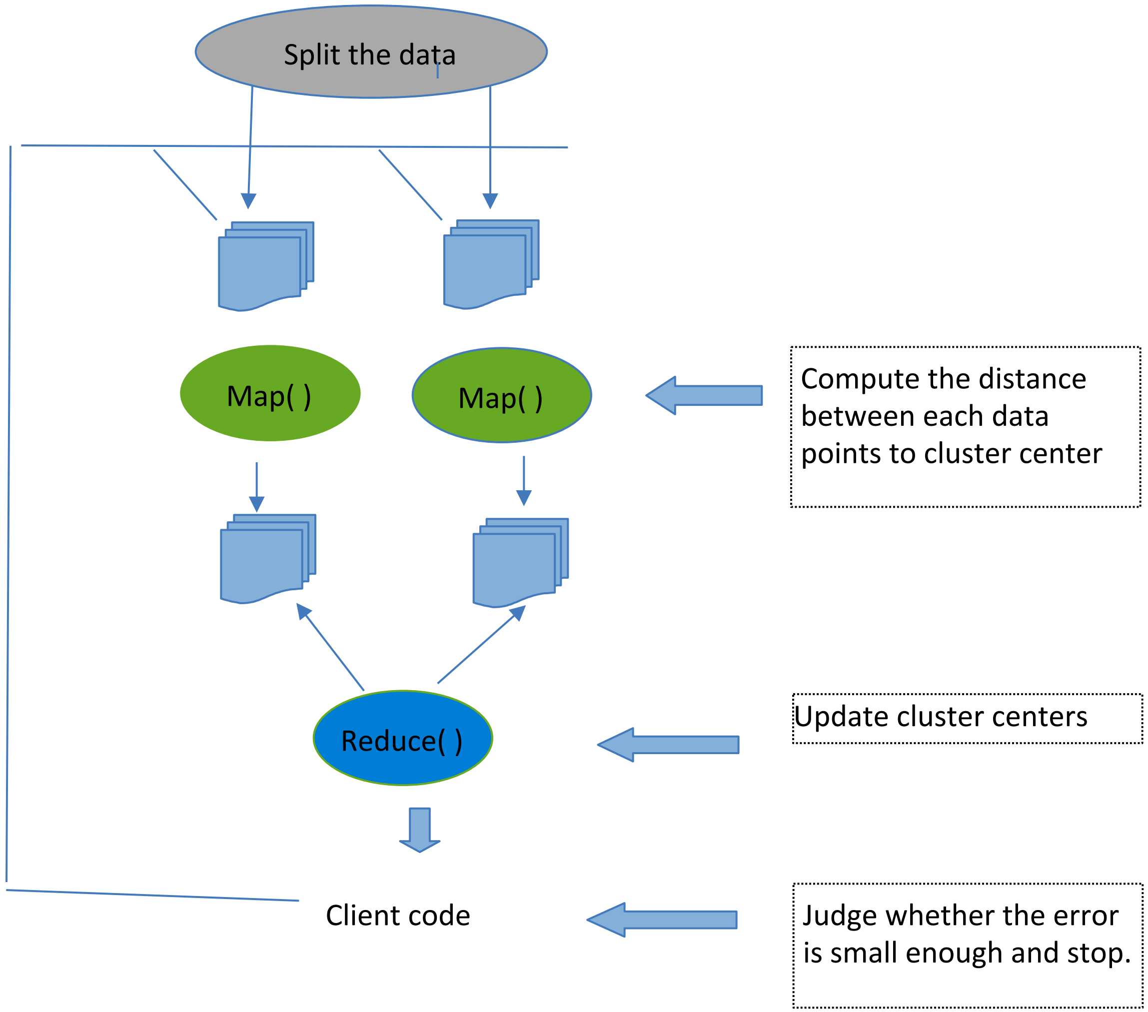
Shortcomings:

1. Offline: It does not do Online data analysis.

2. Batched processing. How about Streaming data? Storm and Spark Streaming are the solution.

3. Not suitable for Data Mining problems based on iterative. In reality, a lot of algorithms are iterative based. That's bad.

Kmeans application with MapReduce framework



Kmeans application with MapReduce

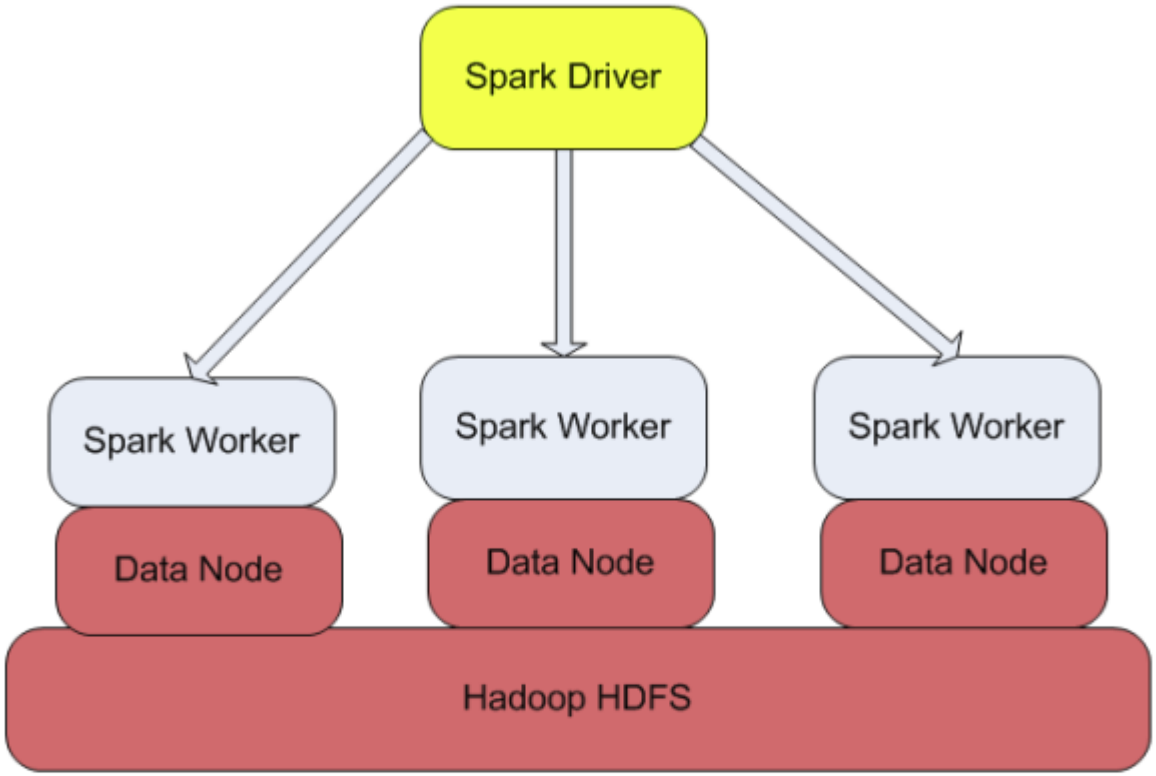
Training set:
Number of data sets: 12 million

Time 6000 s – 8000 s

From a report (Hong Kong University of Science and Technology)

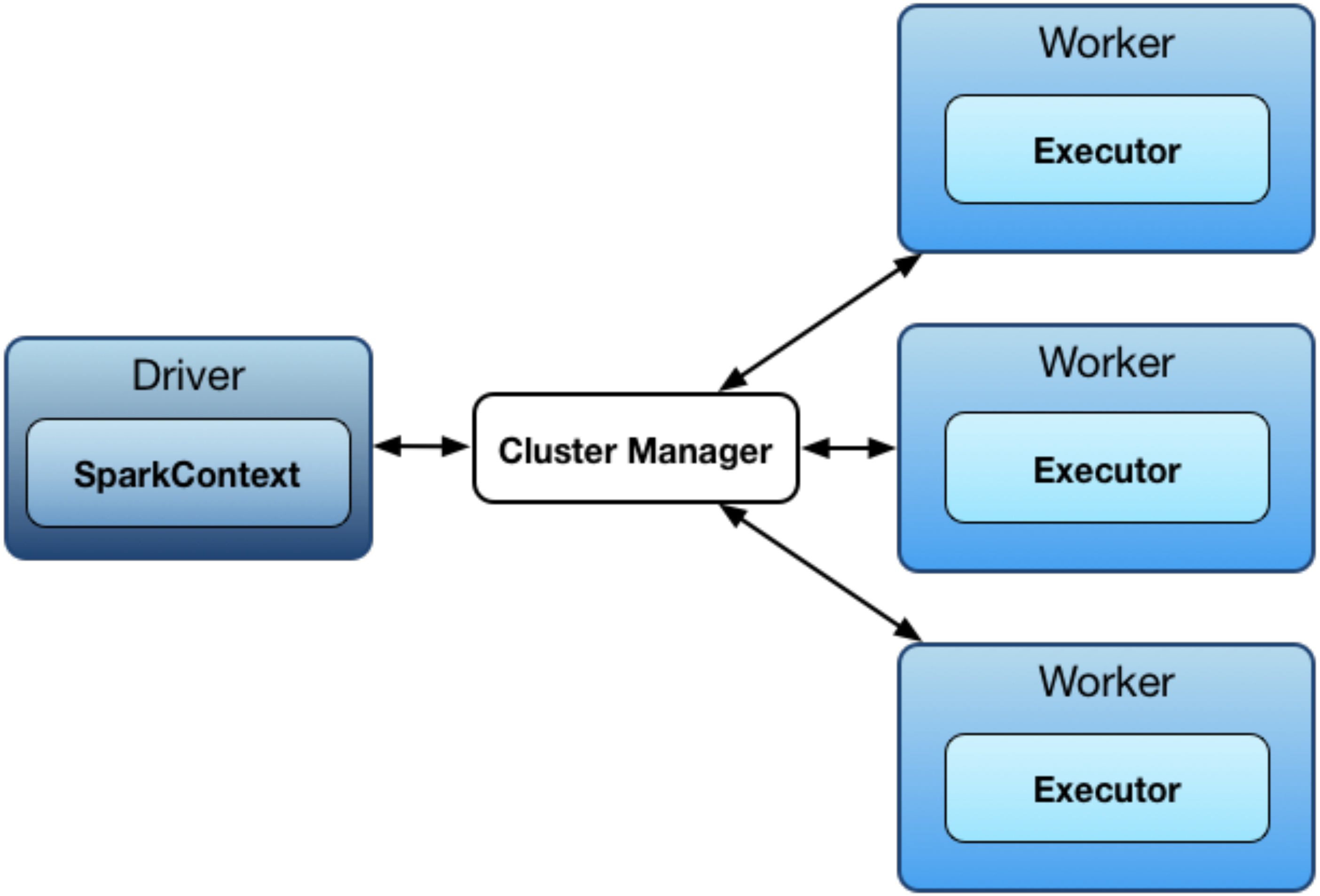
Spark: a big data analysis platform

Architecture



Spark: a big data analysis platform

Architecture



Kernel component of Spark

RDD:

Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

READ-only.

Spark

With a promise of speeds up to 100 times faster than Hadoop MapReduce and comfortable APIs,

The secret is that it runs in-memory on the cluster, and that it isn't tied to Hadoop's MapReduce two-stage paradigm. This makes repeated access to the same data much faster.

Spark outperforms Hadoop?

NO!

Spark performs better when all the data fits in the memory, especially on dedicated clusters; Hadoop MapReduce is designed for data that doesn't fit in the memory and it can run well alongside other services.

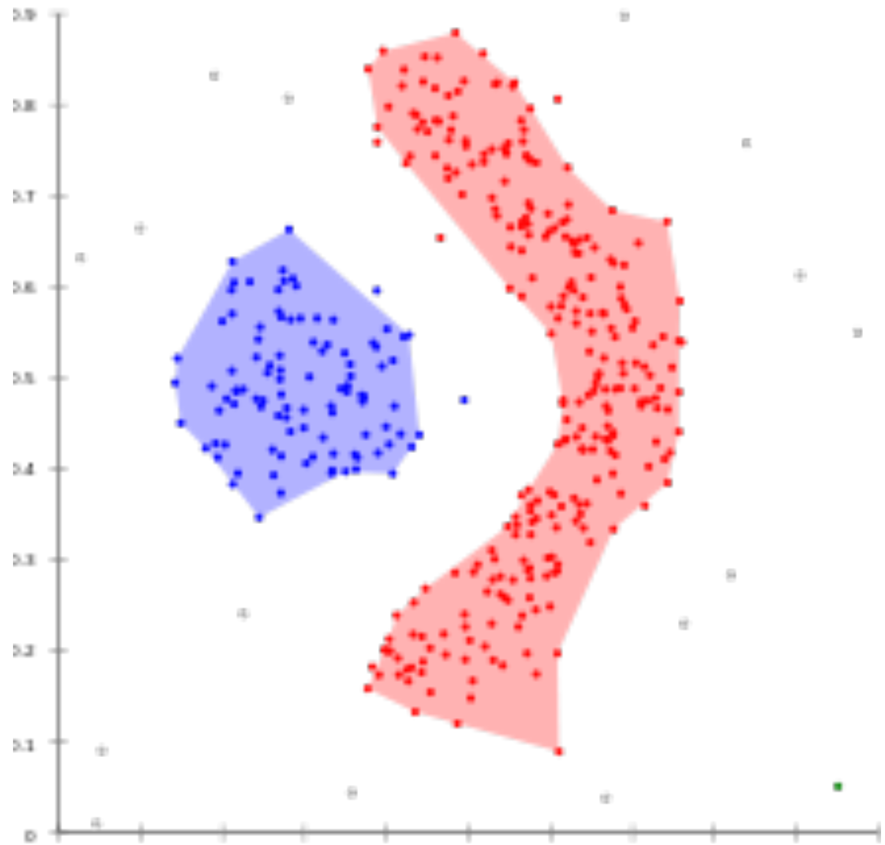
Kmeans application with Spark Mlib

Training set:
Number of data sets: 12 million
Number of features: 500

Storage	47 GB	7 GB
Time	240 s	58 s

One use case: DBSCAN with Spark implementation

DBSCAN algorithm example:



One use case: DBSCAN with Spark implementation

DBSCAN algorithm concepts:

Consider a set of points in some space to be clustered. For the purpose of DBSCAN clustering, the points are classified as core points, (density-)reachable points and noise points, as follows:

- A point p is a core point if at least minPts points are within distance ε of it, and those points are said to be *directly reachable* from p. No points are *directly reachable* from a non-core point.
- A point q is reachable from p if there is a path $p_1, \ldots, p_n$ with $p_1 = p$ and $p_n = q$, where each p_{i+1} is directly reachable from p_i (so all the points on the path must be core points, with the possible exception of q).
- All points not reachable from any other point are outliers (noise).

Algorithm (DBSCAN algorithm)

Input (eps,minpts,D) Output (a set of clusters)

```
initialize all points as unvisited
for each unvisted point p ∈ D do
    mark p as visited
    Let N be eps-neighborhood of p
    if the size of N < minpts points then
        mark p as noise
    else
        create a new cluster C, and add p to C
        for each point p′ ∈ N
            if p′ in unvisited then
                mark p′ as visited
                let N′ be the eps-neighborhood of p′
                if the size of N′ is >= minpts then
                    add those points to N
                endif
            endif
            if p′ is not yet a member of any cluster
                add p′ to C
            endif
        endfor
    endif
endfor
```

One use case: DBSCAN with Spark implementation

Method and Algorithm:

Scalable DBSCAN algorithm with spark:

1. driver reads data file and divides data points into more partitions (p1, p2, p3, p4, for example).

2. each executor compute its points and generate clusters

3. after each executor finishes its task, send partial clusters to driver.

4. driver generates the final clusters.

One use case: DBSCAN with Spark implementation

Experiments and results:

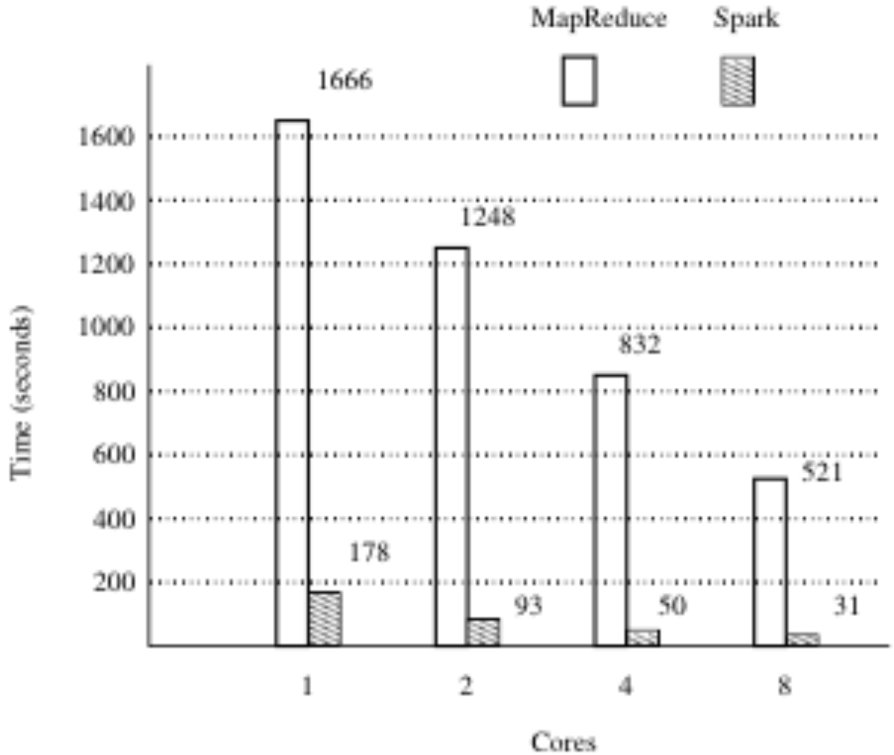


Figure 7: **Time used by MapReduce and Spark.** Number of poinst: 10000, dimension: 10, eps, 25.0, minPnts: 5

Spark VS MapReduce (Hadoop)

Hadoop:Big data and efficient to deal one pass task (most operations on disk).
Shortcoming: intermediate file read and write to the disk.
No communications between mappers.

Spark: Big data and and efficient to deal with iterative algorithms (computation in memory).

Shortcoming: Requirement for memory.

Processing System

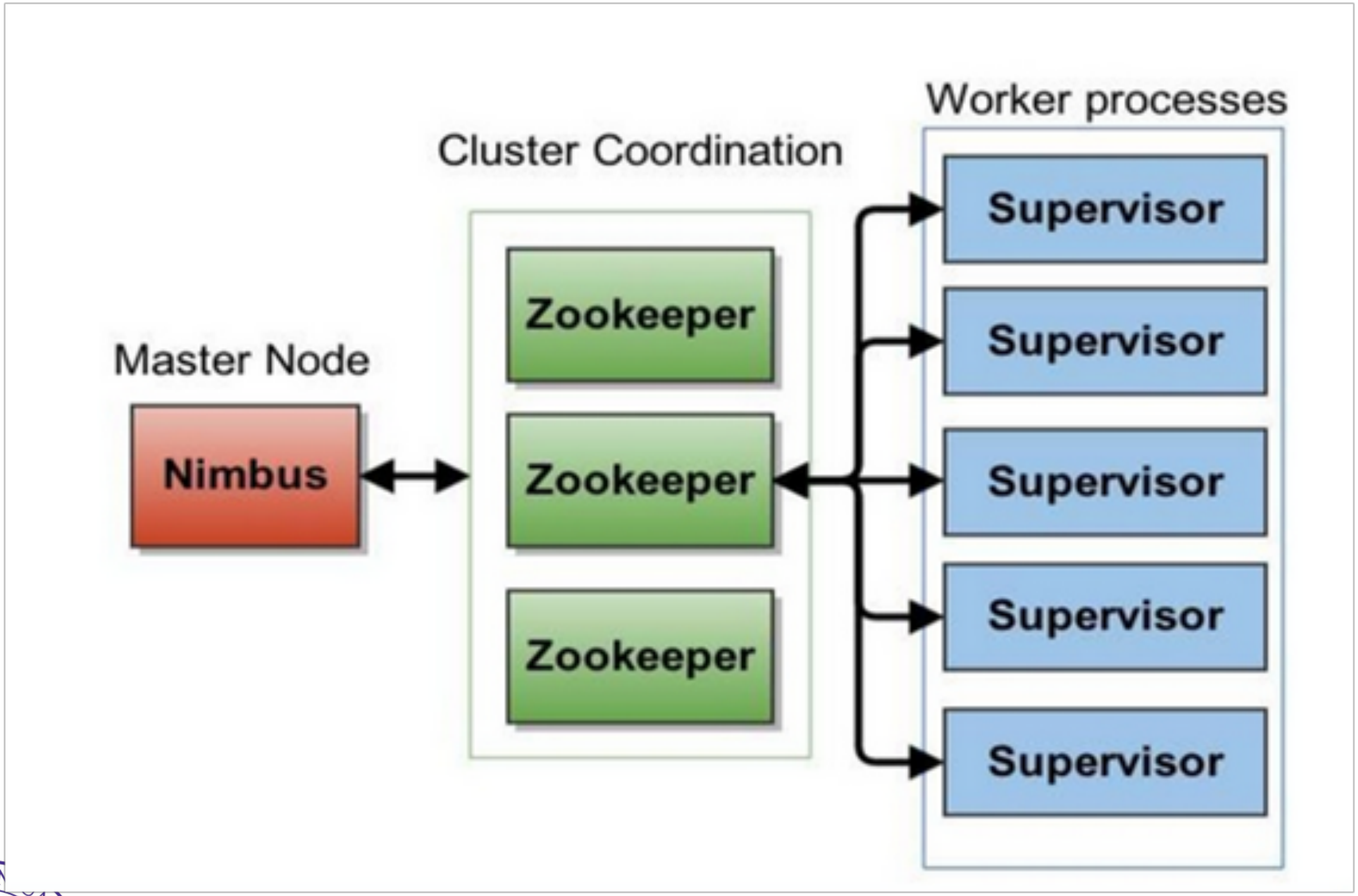
Batch: Examples: Hadoop (MapReduce) ; Spark Core

- Has access to all data
- Might compute something big and complex
- Is generally more concerned with throughput than latency of individual components of the computation
- Has latency measured in minutes or more

Streaming: Storm; Spark Streaming

- Computes a function of one data element, or a smallish window of recent data
- Computes something relatively simple
- Needs to complete each computation in near-real-time -- probably seconds at most
- Computations are generally independent
- Asynchronous - source of data doesn't interact with the stream processing directly, like by waiting for an answer

Storm Architecture



Important Components of Storm

Spout, ***bolt***, and ***topology***.

Topology is like MapReduce job.
One key difference is that MapReduce job eventually finishes.
A topology runs forever.
A topology is a graph of spouts and bolts.

Storm runs 2 tasks: **Spouts** and **Bolts**.

In a topology, **spout** will act as data receiver from external sources and creator of Stream for bolts to process.

Bolts can be chained serially or in parallel.

```
public class WordReader implements IRichSpout {
    private SpoutOutputCollector collector;
    private FileReader fileReader;
    private boolean completed = false;
    private TopologyContext context;
    public boolean isDistributed() {return false;}

    public void ack(Object msgId) {
                System.out.println("OK:"+msgId);
    }

    public void open(Map conf, TopologyContext context, SpoutOutputCollector
    collector) {
            try {
                this.context = context;
                this.fileReader = new FileReader(conf.get("wordsFile").toString());
            } catch (FileNotFoundException e) {
                throw new RuntimeException("Error reading file
    ["+conf.get("wordFile")+"]");
            }
                this.collector = collector;
        }
```

```java
public void nextTuple() {
    if(completed){
       try {
           Thread.sleep(1);
       } catch (InterruptedException e) {
       // print something.
   }
        return;
     }
     String str;
     BufferedReader reader = new BufferedReader(fileReader);
     try{
         while((str = reader.readLine()) != null){
             this.collector.emit(new Values(str));
         }
     }catch(Exception e){
         throw new RuntimeException("Error reading tuple",e);
     }finally{
         completed = true;
     }
  }
```

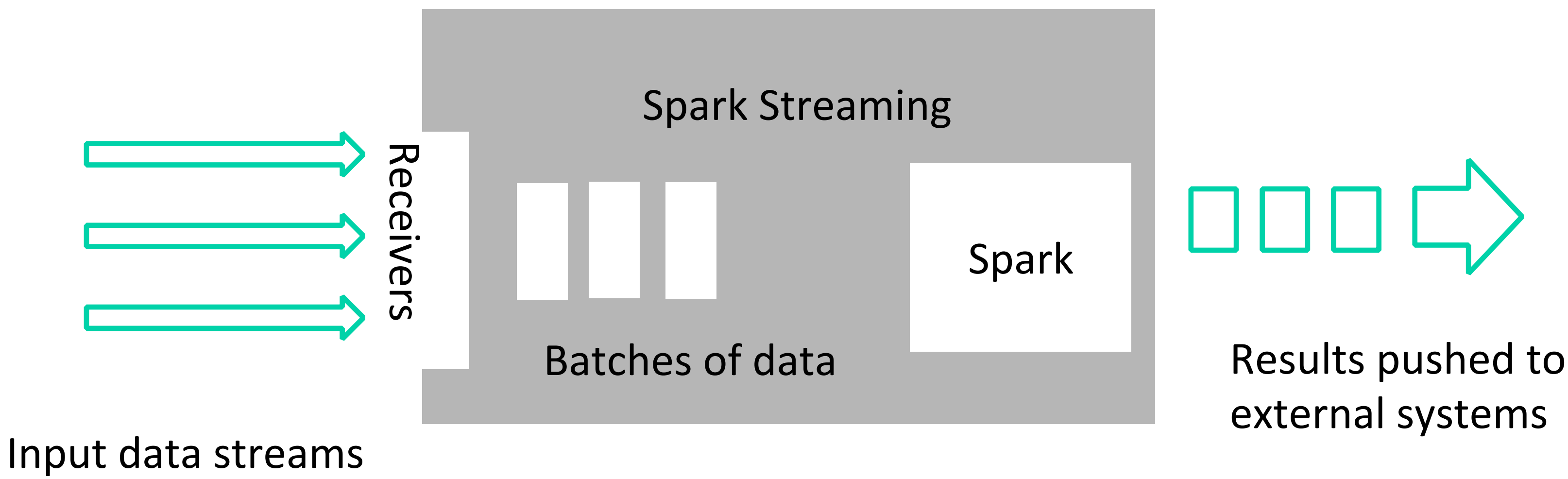
```
public class WordNormalizer implements IRichBolt{
        private OutputCollector collector;
        public void cleanup(){}
        /**
    * *bolt* receive text lines from text file and
          *  normalize them.
    * convert text lines to lowercase and emit.
    */
        public void execute(Tuple input){
            String sentence = input.getString(0);
            String[] words = sentence.split(" ");
            for(String word : words){
                word = word.trim();
                if(!word.isEmpty()){
                    word=word.toLowerCase();
                    //emit this word
                    List a = new ArrayList();
                    a.add(input);
                    collector.emit(a,new Values(word));
                }
            }
            //response of tuple
    collector.ack(input);
        }
```

```
import spouts.WordReader;
import backtype.storm.Config;
import backtype.storm.LocalCluster;
import backtype.storm.topology.TopologyBuilder;
import backtype.storm.tuple.Fields;
import bolts.WordCounter;
import bolts.WordNormalizer;

public class TopologyMain {
   public static void main(String[] args) throws
InterruptedException {
       //define topologybuilder
   TopologyBuilder builder = new TopologyBuilder());
       builder.setSpout("word-reader", new WordReader());
       builder.setBolt("word-normalizer", new
WordNormalizer()).shuffleGrouping("word-reader");
        …
```

Link spout and bolts Using shuffleGrouping

Spark Stream Architecture and Abstraction



High-level architecture of spark streaming

Learning spark: lightning fast data analysis

Example of Spark Streaming:

```
// create a streamingcontext with a 1-second batch size from a SparkConf
JavaStreamingContext jssc= new JavaStreamingContext(conf, Durations.seconds(1));
// create a Dstream from all the input on port 7777
JavaDStream<String> lines = jssc.socketTextStream(“localhost”, 7777);
// filter our Dstream for lines with “error”
JavaDStream<String> errorlines = lines.filter(new Function<String, Boolean>() {
   public Boolean call(String line) {
     return line.contains(“error”);
}});
// Print out the lines with errors
errorlines.print();

Jssc.start();
Jssc.awaitTermination();
```

Learning spark: lightning fast data analysis

Input sources

1. Stream of files:
2. JavaDStream<String> logData = jssc.textFileStream(logsDirectory);

3. Stream of network:
4. JavaDStream<String> lines = jssc.socketTextStream(“localhost”, 7777);

5. Additional sources:
6. Apache Kafka, Apache Flume, etc.

Spark Streaming VS Storm

Language Options

Core Storm	Spark Streaming
Java	Java
Clojure	Scala
Scala	Python
Python	
Ruby	
Others	

http://www.slideshare.net/ptgoetz/apache-storm-vs-spark-streaming

Spark Streaming VS Storm

Programming Model

	Core Storm	Spark Streaming
Stream primitive	Tuple	Dstream
Stream Source	Spouts	HDFS, Network
Transformation	Bolts	Transformation
Output/Persistence	Bolts	Print(), saveasTextFiles()

http://www.slideshare.net/ptgoetz/apache-storm-vs-spark-streaming

Spark Streaming VS Storm

Reliability Model

	Core Storm	Spark Streaming
At Most Once	Yes	No
At least Once	Yes	No*
Exactly Once	Yes	Yes

http://www.slideshare.net/ptgoetz/apache-storm-vs-spark-streaming

Spark Streaming VS Storm

Reliability Limitations: Apache Storm

Exactly once processing requires a durable data source.
At least once processing requires a reliable data source.
With durable and reliable sources, Storm will not drop data.

http://www.slideshare.net/ptgoetz/apache-storm-vs-spark-streaming

Spark Streaming VS Storm

Reliability Limitations: Spark Streaming

Fault tolerance and reliability require HDFS-backed data source. Checkpointing.

Network data sources (Kafka, etc.) are vulnerable to data loss in the event of a worker node failure.

http://www.slideshare.net/ptgoetz/apache-storm-vs-spark-streaming

Performance

Spark Streaming's Java or Scala-based execution architecture is claimed to be 4X to 8X faster than Apache Storm using the WordCount benchmark.

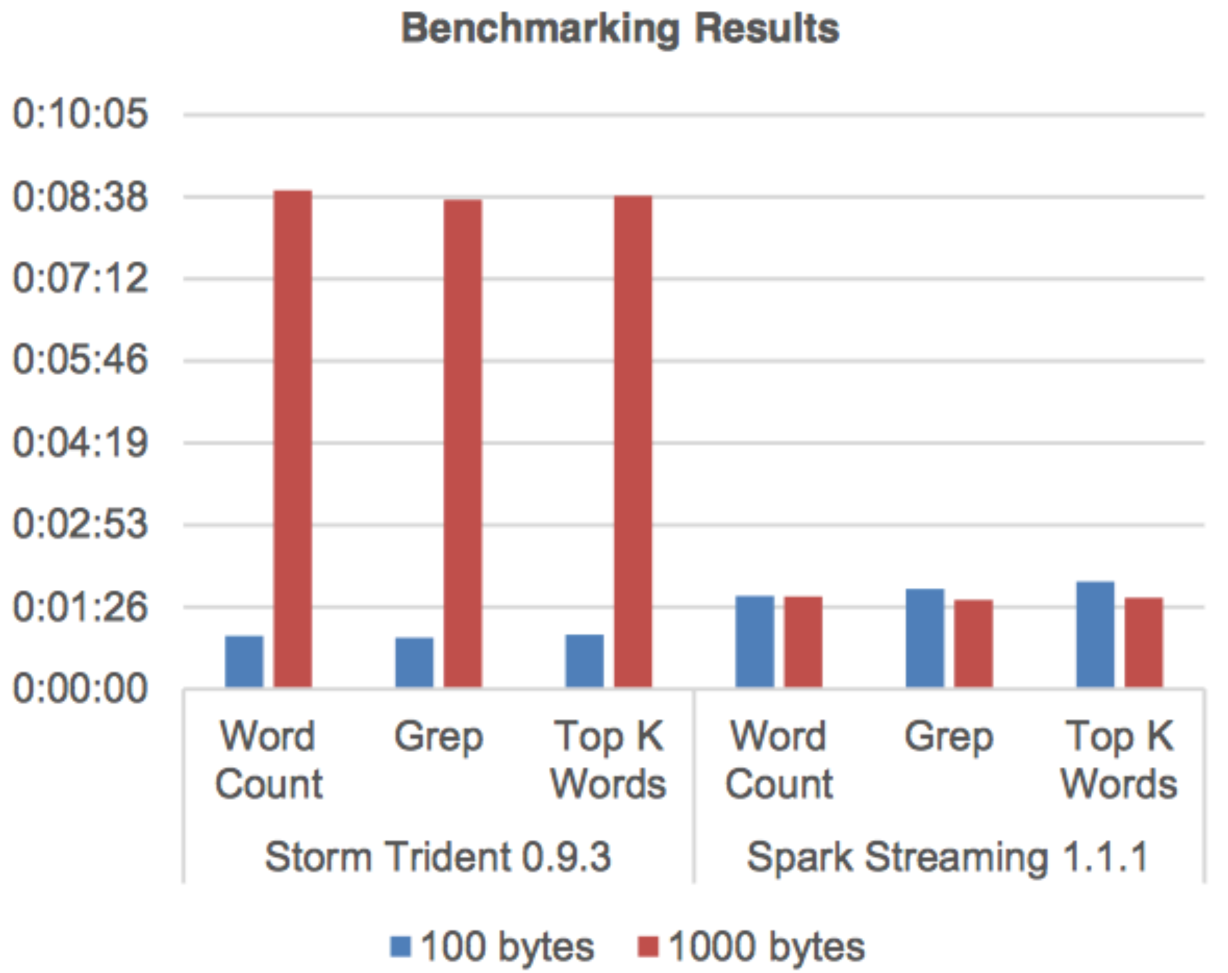


Fig. 3. Time taken by Storm Trident and Spark Streaming to process one million rows in the architecture already proposed in Section V.

http://www.cs.toronto.edu/~patricio/docs/Analysis_of_Real_Time_Stream_Processing_Systems_Considering_Latency.pdf

Performance

The main conclusion of this section was that Storm was around 40% faster than Spark, processing tuples of small size (around the size of a tweet). However, as the tuple's size increased, Spark had better performance maintaining the processing times.

Questions? Or Comments?