Anton Fagerberg

anton.fagerberg@jayway.com

Spark vs Hadoop MapReduce

Word count

The "Hello World" of MapReduce

input:

```
My Bonnie lies over the ocean
My Bonnie lies over the sea
My Bonnie lies over the ocean
Oh, bring back my Bonnie to me...
```

output:

```
Bonine, 4
lies, 3
...

to, 1
```

MapReduce

Simple concept

Map: transform things

Reduce: combine things

Map - transform things

```
My Bonnie lies over the ocean
My Bonnie lies over the sea
My Bonnie lies over the ocean
Oh, bring back my Bonnie to me...
```

```
(My, 1)
(Bonnie, 1)
(lies, 1)

...

(Bonnie, 1)
(to, 1)
(me, 1)
```

Reduce - combine things

Input:

```
(My, 1)
(Bonnie, 1)
(lies, 1)

...

(Bonnie, 1)
(to, 1)
(me, 1)
```

Output:

```
(My, 3)
(Bonnie, 4)
(lies, 3)

...

(me, 1)
```

Hadoop MapReduce

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.net.URI;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFor
import org.apache.hadoop.mapreduce.lib.output.FileOutputF
import org.apache.hadoop.mapreduce.Counter;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.StringUtils;
```

Map class

```
public class WordCount2 {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    static enum CountersEnum { INPUT_WORDS }

    private final static
      IntWritable one = new IntWritable(1);
    private Text word = new Text();

    private boolean caseSensitive;
    private Set<String> patternsToSkip =
    new HashSet<String>();

    private Configuration conf;
    private BufferedReader fis;
```

- Object
- IntWritable

```
@Override
public void setup(Context context) throws IOException,
    InterruptedException {
  conf = context.getConfiguration();
  caseSensitive = conf.getBoolean("wordcount.case.sensiti
  if (conf.getBoolean("wordcount.skip.patterns", true)) {
    URI[] patternsURIs = Job.getInstance(conf).getCacheFi
    for (URI patternsURI : patternsURIs) {
      Path patternsPath = new Path(patternsURI.getPath())
      String patternsFileName = patternsPath.getName().to
      parseSkipFile(patternsFileName);
    }
  }
}
```

```
private void parseSkipFile(String fileName) {
  try {
    fis = new BufferedReader(new FileReader(fileName));
    String pattern = null;
    while ((pattern = fis.readLine()) != null) {
      patternsToSkip.add(pattern);
    }
  } catch (IOException ioe) {
    System.err.println("Caught exception while parsing th
        + StringUtils.stringifyException(ioe));
  }
}
```

- null checks
- try-catch

Map

```
@Override
public void map(Object key, Text value, Context context
                ) throws IOException, InterruptedExceptio
  String line = (caseSensitive) ?
      value.toString() : value.toString().toLowerCase();
  for (String pattern : patternsToSkip) {
    line = line.replaceAll(pattern, "");
  }
  StringTokenizer itr = new StringTokenizer(line);
  while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    context.write(word, one); // hmmm "one"
    Counter counter = context.getCounter(CountersEnum.cla
        CountersEnum.INPUT_WORDS.toString());
    counter.increment(1);
  }
}
}
```

Reduce

```
public static class IntSumReducer
      extends Reducer<Text,IntWritable,Text,IntWritable>
   private IntWritable result = new IntWritable();

   public void reduce(Text key, Iterable<IntWritable> va
                      Context context
                      ) throws IOException, InterruptedE
     int sum = 0;
     for (IntWritable val : values) {
       sum += val.get(); // Hmmmm...
     }
     result.set(sum); // Hmmmm...
     context.write(key, result);
   }
 }
```

```
Configuration conf = new Configuration();
GenericOptionsParser optionParser = new GenericOptionsPa
String[] remainingArgs = optionParser.getRemainingArgs(
if (!(remainingArgs.length != 2 | | remainingArgs.lengt
  System.err.println("Usage: wordcount <in> <out> [-ski
  System.exit(2);
}
Job job = Job.getInstance(conf, "word count");
job.setJarByClass(WordCount2.class);
job.setMapperClass(TokenizerMapper.class); // Hmmmm...
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

List<String> otherArgs = new ArrayList<String>();
for (int i=0; i < remainingArgs.length; ++i) {
  if ("-skip".equals(remainingArgs[i])) {
    job.addCacheFile(new Path(remainingArgs[++i]).toUri
    job.getConfiguration().setBoolean("wordcount.skip.pa
  } else {
    otherArgs.add(remainingArgs[i]);
  }
}
FileInputFormat.addInputPath(job, new Path(otherArgs.ge
FileOutputFormat.setOutputPath(job, new Path(otherArgs.
```

Spark

Spark is 100 times faster than Hadoop

APACHE Spark™ *Lightning-fast cluster computing*

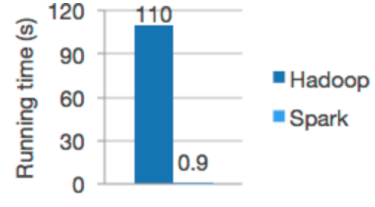
Download | Libraries | Documentation | Examples | Community | FAQ | Apache Software Foundation

Apache Spark™ is a fast and general engine for large-scale data processing.

Speed

Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.

Apache Spark has an advanced DAG execution engine that supports cyclic data flow and in-memory computing.



Logistic regression in Hadoop and Spark

Latest News

Spark 2.0.0 released (Jul 26, 2016)

Spark 1.6.2 released (Jun 25, 2016)

Call for Presentations for Spark Summit EU is Open (Jun 16, 2016)

Preview release of Spark 2.0 (May 26, 2016)

Archive

Download Spark

Or...?

Daytona GraySort contest: 100 TB

Hadoop (2013)

72 minutes

Spark (2014)

23 minutes

Hadoop

- Dedicated data center
- 2100 nodes
- 50,400 cores
- Rate per node: 0.67 GB/min

Spark

- Amazon EC2 (i2.8xlarge)
- 206 nodes
- 6,592 cores
- Rate per node: 20.7 GB/min

https://databricks.com/blog/2014/10/10/spark-petabyte-sort.html

Presentation overview

- What is Spark?
- How do we use Spark?
- How does Spark *really* work?

What is Spark?

Apache Spark is a fast and general engine for large-scale data processing

Distributed programming is the art of solving the same problem that you can solve on a single computer using multiple computers.

Mikito Takada, Distributed systems for fun and profit

Powered by Spark

- Spark SQL
- Spark Streaming
- MLib
- GraphX

Runs on

- Hadoop (YARN)
- Mesos
- Standalone
- (EC2)

Access data on

- HDFS
- Cassandra
- HBase
- S3
- Hive
- Tachyon
- or any Hadoop data source...

Languages

- Scala
- Java
- Python
- R

How do we use Spark?

Higher order functions

Normal function

Scala:

```scala
def helloNumber(number: Int): String = {
  s"Hello $number"
}
```

Java:

```java
public String helloNumber(int number) {
  return "Hello " + number;
}
```

Python

```python
def helloNumber(number):
        return "Hello {0}".format(number)
```

A higher-order function

takes one or more functions as arguments.

Map

Map applies a function to every element in a collection:

```
List(1, 2, 3).map(helloNumber)

List(helloNumber(1), helloNumber(2), helloNumber(3))

List("Hello 1", "Hello 2", "Hello 3")
```

Map

Map applies a function to every element in a collection:

```
List(1, 2, 3).map(nr => nr + 1)

List(2, 3, 4)
```

Alternative syntax

```
List(1, 2, 3).map(_ + 1)

List((1 + 1), (2 + 1), (3 + 1))

List(2, 3, 4)
```

Filter

```
List(1, 2, 3, 4).filter(nr => nr % 2 == 0)

List(2, 4)
```

```
List(1, 2, 3, 4).filter(_ > 2)

List(3, 4)
```

Map

```
List(1, 2, 3).map(nr => List(nr, nr))

List(List(1, 1), List(2, 2), List(3, 3))
```

FlatMap

```
List(1, 2, 3).flatMap(nr => List(nr, nr))

List(1, 1, 2, 2, 3, 3)
```

FlatMap

```
val sentences = List("hello world", "how are you")

sentences.flatMap(line => line.split(' '))


List("hello", "world", "how", "are", "you")
```

Reduce

```
List(1, 2, 3).reduce(_ + _)

1 + 2 + 3

6
```

Reduce

```
List(4, 5, 6).reduce(_ + _)
```

Reduce (complicated)

```
List(1, 2, 3).reduce { (acc, nr) =>
  println(s"$acc = acc, $nr = nr")
  acc + nr
}
```

output:

```
1 = acc, 2 = nr
3 = acc, 3 = nr
res1: Int = 6
```

```
List(1, 2, 3, 4)
  .filter(_ % 2 == 0)  // List(2, 4)
  .map(_ * 2)          // List(4, 8)
  .reduce(_ + _)       // 12
```

Normal Scala

```
data
  .filter(n => n % 2 == 0)
  .map(n => n * 2)
  .reduce((a, b) => a + b)
```

Spark (Scala)

```
data
  .filter(n => n % 2 == 0)
  .map(n => n * 2)
  .reduce((a, b) => a + b)
```

Spark (Java)

```
data
  .filter(n -> n % 2 == 0)
  .map(n -> n * 2)
  .reduce((a, b) -> a + b)
```

Spark (Python)

```
data
  .filter(lambda n: n % 2 == 0)
  .map(lambda n: b * 2)
  .reduce(lambda a, b: a + b)
```

Spark (Java old-school)

```
lines.map(new Function<String, Integer>() {
  public Integer call(String s) { return s.length(); }
});
```

RDD (Resilient Distributed Datasets)

- Fault-tolerant collection of elements
- Can be operated on in parallel
- Lazy
- Immutable

```
map[B](f: A => B): List[B]
map[B](f: A => B): RDD[B]
```

```
flatMap[B](f: A => TraversableOnce[B]): List[B]
flatMap[B](f: A => TraversableOnce[B]): RDD[B]
```

```
filter(pred: A => Boolean): List[A]
filter(pred: A => Boolean): RDD[A]
```

```
reduce(op: (A, A) => A): A
reduce(op: (A, A) => A): A
```

```
fold(z: A)(op: (A, A) => A): A
fold(z: A)(op: (A, A) => A): A
```

```
aggregate[B](z:=>B)(seqop:(B,A) => B,combop:(B,B)=>B):B
aggregate[B](z:  B)(seqop:(B,A) => B,combop:(B,B)=>B):B
```

data.txt:

```
hello
world!
```

code:

```
val lines = sc.textFile("data.txt")
// RDD[String]
// RDD("hello", "world!")

val lineLengths = lines.map(s => s.length)
// RDD[Int]
// RDD(5, 6)

val totalLength = lineLengths.reduce((a, b) => a + b)
// Int
// 11
```

Pair RDD

- RDD containing a Tuple2 (Key, Value)

Scala:

```
Tuple2("hello", 1)

("hello", 1)

"hello" -> 1
```

Pair RDD functions

- Joins
- ByKey-operations

Joins

```
val names =
  sc.parallelize(List(
    (1, "Alice"),
    (2, "Bob")
  ))

val ages =
  sc.parallelize(List(
    (1, 28)
  ))

names.join(profile)
// RDD( (1, ("Alice", 28)) )


names.leftOuterJoin(ages)
// RDD( (1, ("Alice", Some(28)), (2, ("Bob", None))) )
```

(fullOuterJoin, rightOuterJoin, coGroup and so on...)

Demo

Word count (slowly)

Spark UI

Lazy evaluation

How does it *really* work?

- A **Spark application** consists of one or more **jobs**.
- A **job** consists one or more **stages**.
- A **stage** is divided into one or more **tasks**.
- A **task** processes one **partition**.

Spark Job

- **Input: Value -> RDD**
  - Convert local value
  - Read from file / HDFS / ...
- **Transformation(s): RDD -> RDD**
  - `map` / `filter` / `flatMap` / ...
  - Usually many transformations in one job
  - Lazy
- **Action**
  - Returns a value
  - Save to file / HDFS / ...
  - Eager

Spark job

```
// Read input from file (create RDD)
// lines: RDD[String]
val lines = sc.textFile("data.txt");


// lineLengths: RDD[Int] (transform RDD)
val lineLengths = lines.map(s => s.length());


// Converts RDD to a value
// totalLength: Int
val totalLength = lineLengths.reduce((a, b) => a + b);
```

Spark job

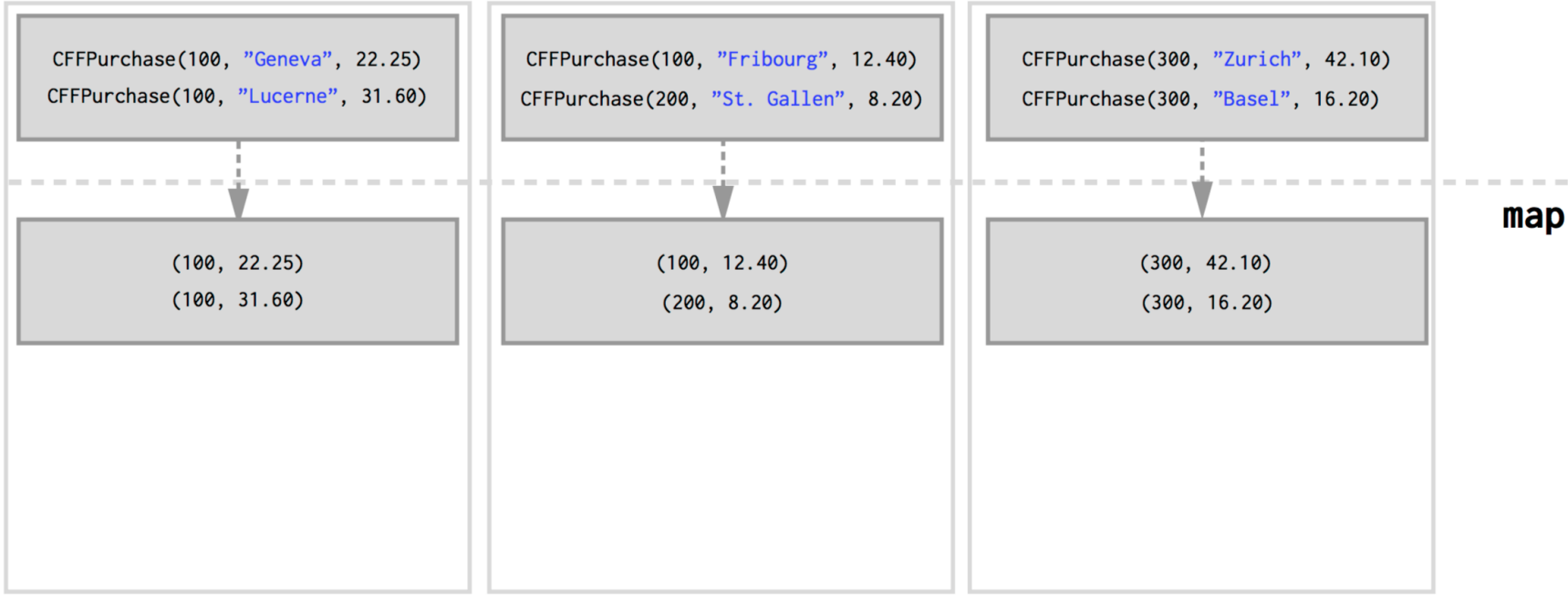
```
data
  .map(...)
  .flatMap(...)
  .filter(...)
  .join(...)
  .reduceByKey(...)
  //...
  .saveAsText(...)
```

(You don't have to write "one-liners")

Stage

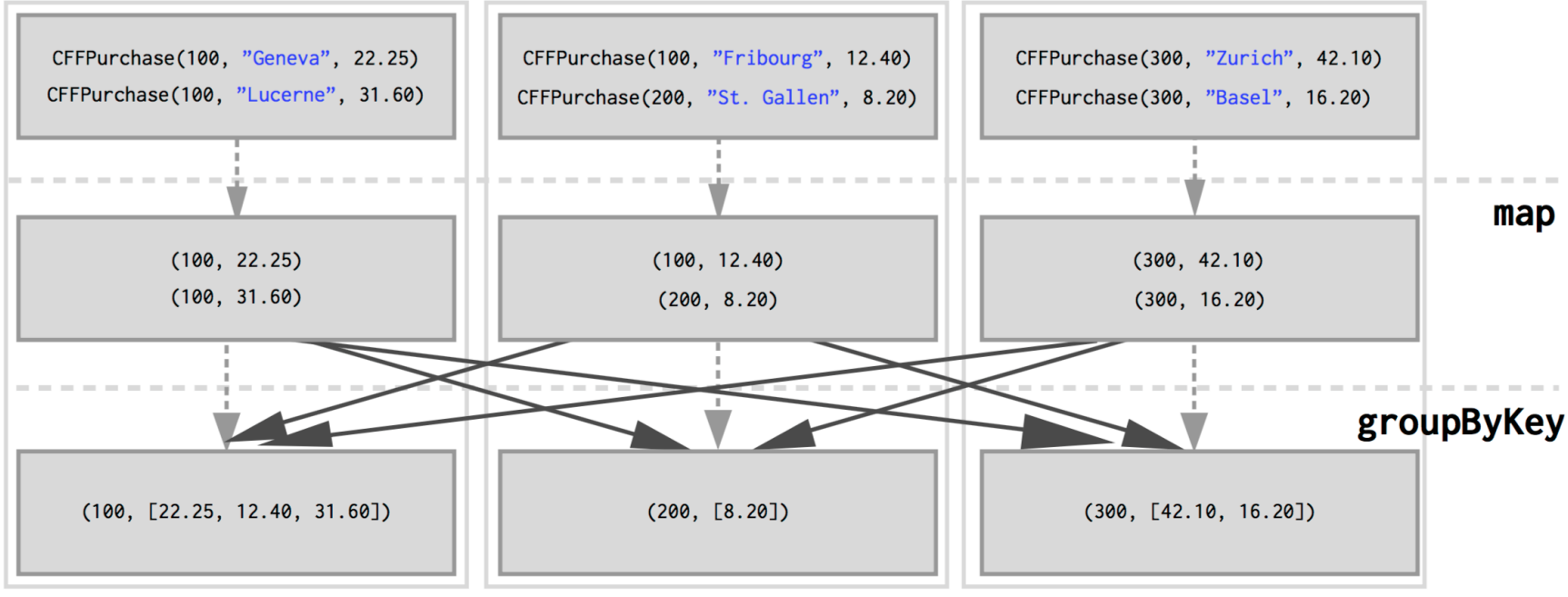
Separated by shuffle operations

Map



http://heather.miller.am/teaching/cs212/slides/week20.pdf

Shuffle



http://heather.miller.am/teaching/cs212/slides/week20.pdf

Shuffle

- Expensive operation:
  - Disk I/O
  - Data serialization
  - Network I/O

- A **job** consists one or more **stages**.
- A **stage** is divided into one or more **tasks**.
- A **task** processes one **partition**.

Partition

- Part, or slice, of the whole data.
- Elements in a partition are on the same machine.

Partitioner

- Assign each object to one partition.
- HashPartitioner / RangePartitioner.
- Custom partitioner.

HashPartitioner

- Given `X` number partitions.
- Object `Y` will end up in partition `Y.hashCode % X`.

HashPartitioner example

Two partitions

```
val items = List("hello", "how", "are", "you", "?")

val hashCodes = items.map(_.hashCode)
// => List(99162322, 103504, 96852, 119839, 63)

val partitions = hashCodes.map(_ % 2)
// => List(0, 0, 0, 1, 1)
```

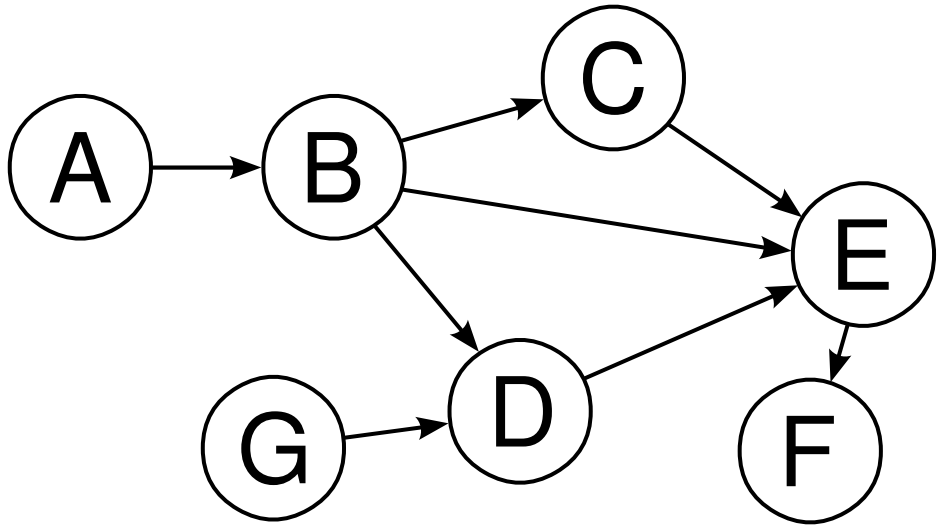
Result

- Partition 0: `"hello"`, `"how"`, `"are"`
- Partition 1: `"you"`, `"?"`

Spark is 100 times faster than Hadoop MapReduce

Apache Spark has an advanced DAG execution engine that supports cyclic data flow and in-memory computing

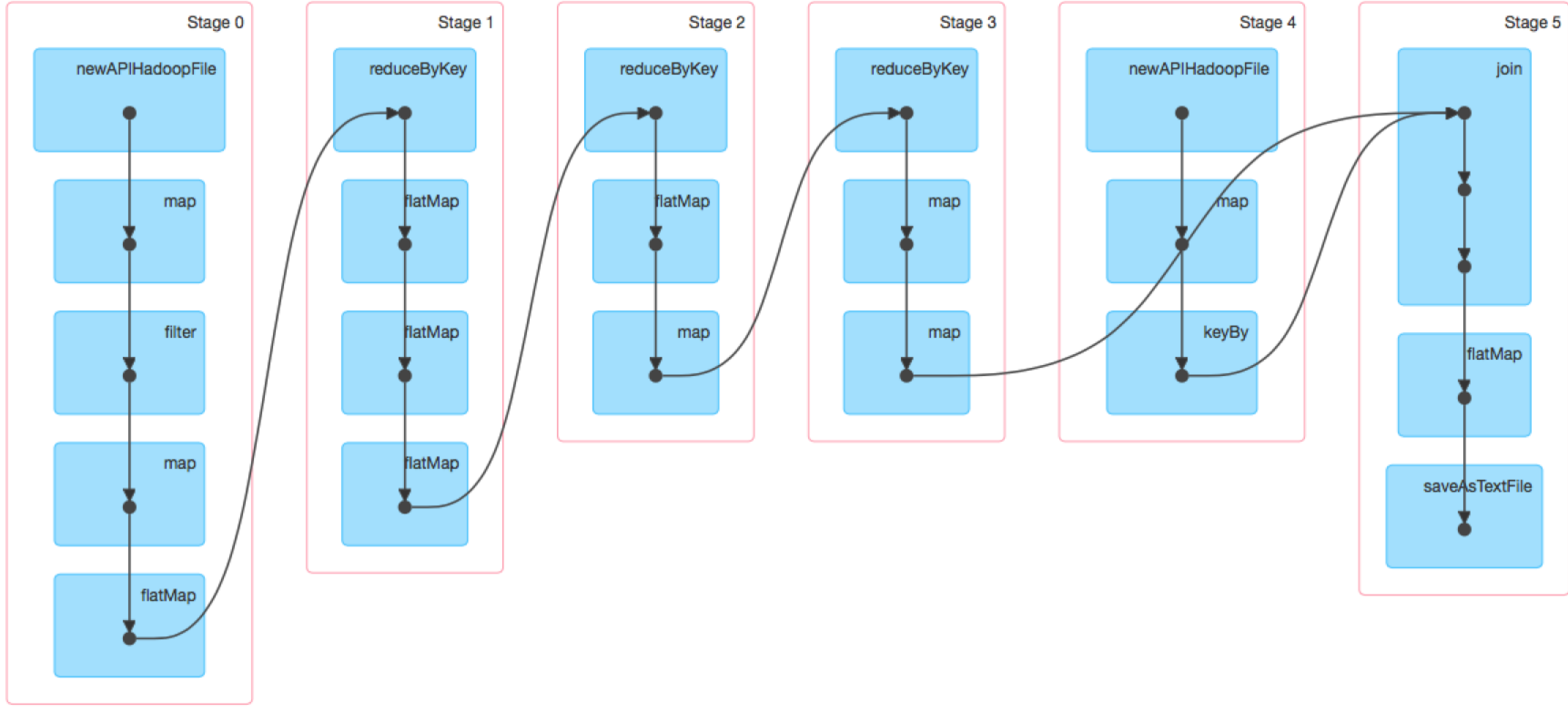
Directed Acyclic Graph (DAG)



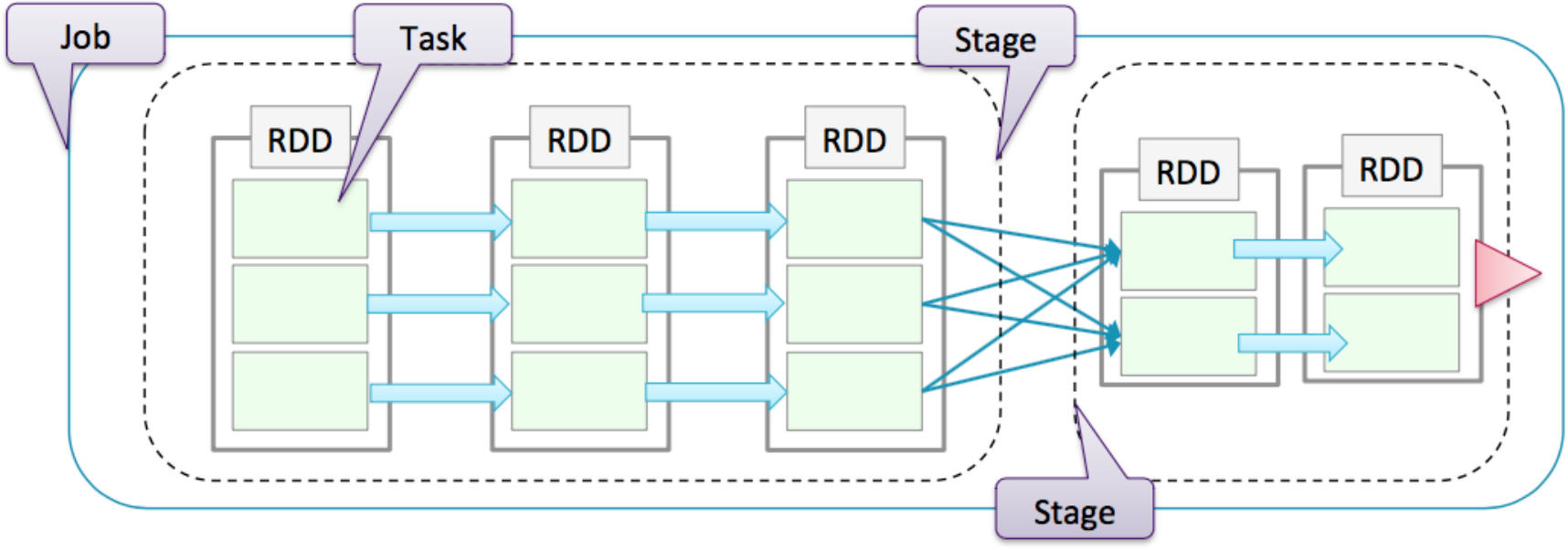
Spark UI

Word count

DAG for a job (Spark UI)



Overview



- 100 times faster than Hadoop MapReduce?
- Transform - single function
- **Resilient** Distributed Dataset (RDD)

Shared variables

Don't do this!

```
var counter = 0
var rdd = sc.parallelize(List(1, 2, 3))

rdd.foreach(x => counter += x)

println("Counter value: " + counter)
```

Shared variables

- Broadcast variables
- Accumulators

Broadcast variables

- Read-only value
- Cached on each machine

Broadcast variables

```
// Convert value to broadcast variable
val broadcastVar = sc.broadcast(Array(1, 2, 3))

// Get value from broadcast variable
sc
.parallelize(Array(1, 2, 3, 4))
.filter(x => broadcastVar.value.contains(x))
```

Accumulators

- Variables that are "added"
  - Associative: `(1 + 2) + 3 == 1 + (2 + 3)`
  - Commutative: `1 + 2 == 2 + 1`
- Predefined (long accumulator)
- Define your own

Example: long accumulator

```
// sc = spark context
val accum = sc.longAccumulator("My Accumulator")


sc
.parallelize(Array(1, 2, 3, 4))
.foreach(x => accum.add(x))


accum.value
// Long = 10
```

If we have time:

- More Spark UI
- Driver
- Cache / persist

Questions?