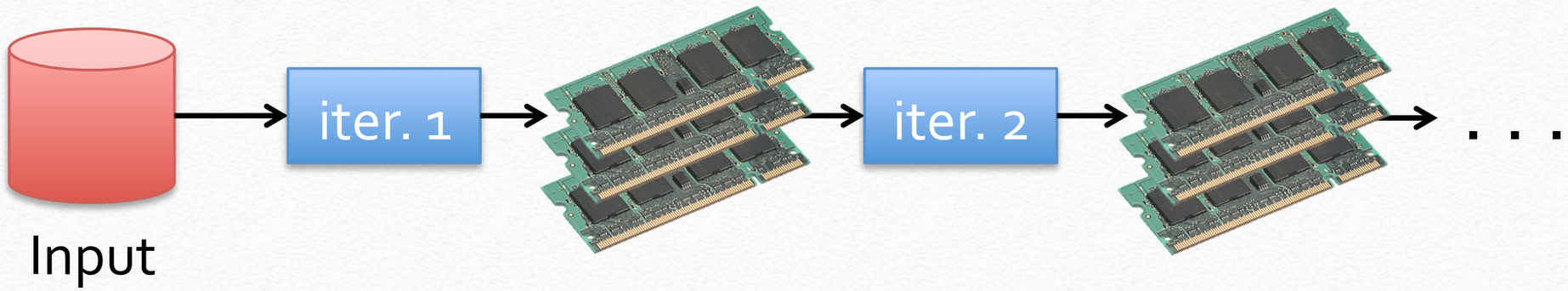
High-Speed In-Memory Analytics over Hadoop and Hive Data

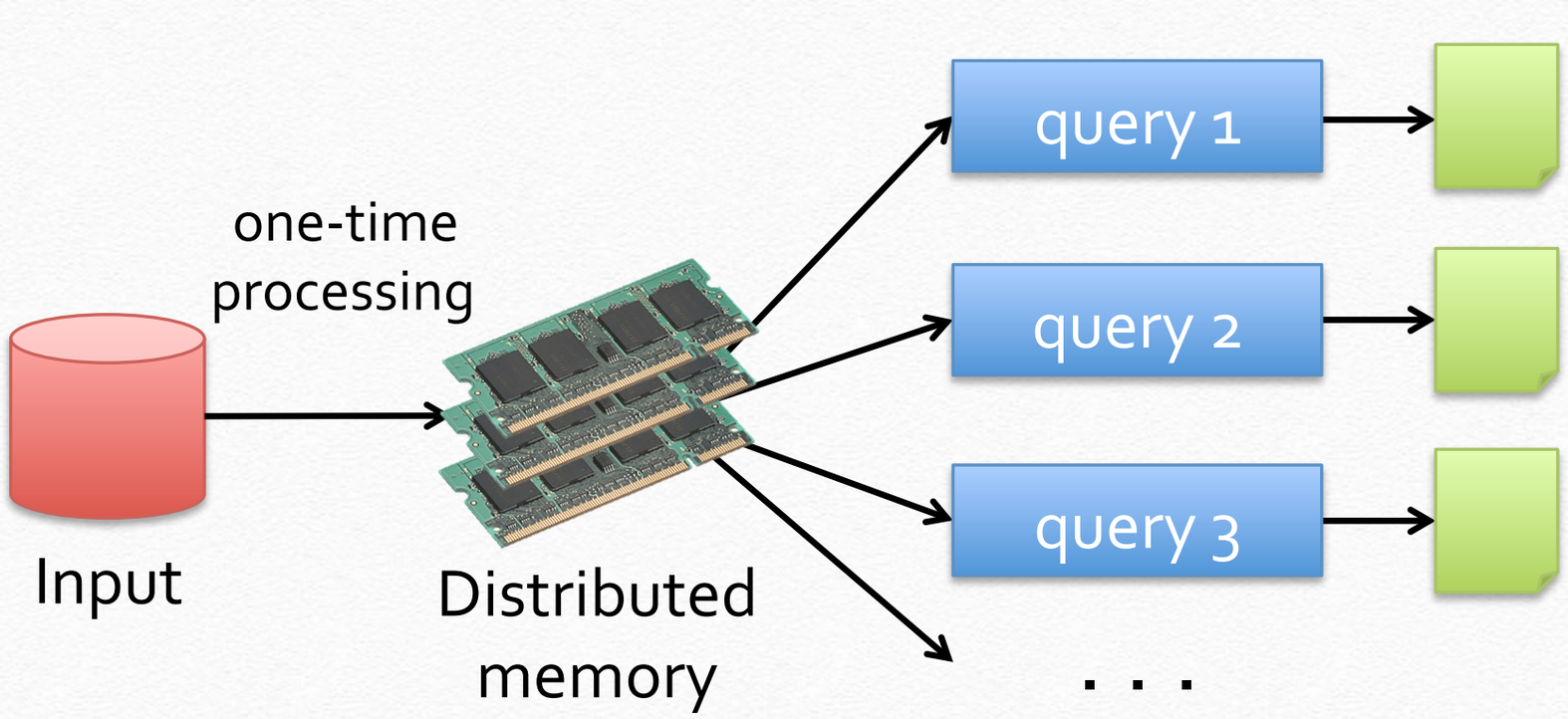
20/04/2016 - Big Data 2016

Apache Spark

- **Not** a modified version of **Hadoop**
- Separate, fast, **MapReduce-like** engine
  - ☑ **In-memory** data storage for very fast iterative queries
  - ☑ General **execution graphs** and powerful optimizations
  - ☑ Up to **40x faster** than Hadoop
- **Compatible** with Hadoop's storage APIs
  - ☑ Can read/write to any Hadoop-supported system, including HDFS, HBase, SequenceFiles, etc

Apache Spark





10-100× faster than network and disk

Spark

Users

CONVIVA

foursquare

quantiFind

KLOUT

YAHOO! RESEARCH

University of California Berkeley

PRINCETON UNIVERSITY

UCSF

Spark Configuration

- Download a **binary release** of **apache Spark:**

- **spark-1.6.1-bin-hadoop2.4.tgz**

Download Spark

The latest release of Spark is Spark 1.3.1, released on April 17, 2015 (release notes) (git tag)

1. Chose a Spark release: 1.3.1 (Apr 17 2015)
2. Chose a package type: Pre-built for Hadoop 2.6 and later
3. Chose a download type: Direct Download
4. Download Spark: spark-1.3.1-bin-hadoop2.6.tgz
5. Verify this release using the 1.3.1 signatures and checksums.

Spark Running

- Running Spark Shell [**scala**]:

```
$:~spark-*/bin/spark-shell
```

- Running Spark Shell [**python**]:

```
$:~spark-*/bin/pyspark
```

- **Spark Shell - Scala**

```
Welcome to

      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.3.1
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_05)

Type in expressions to have them evaluated.

scala>
```

Spark Self-contained applications

- **Java Spark API**

```
import org.apache.spark.api.java.*;

import org.apache.spark.SparkConf;

import org.apache.spark.api.java.function.Function;

public class SimpleApp {

  public static void main(String[] args) {

    String logFile = "YOUR_SPARK_HOME/README.md"; // Should be some file on your system

    SparkConf conf = new SparkConf().setAppName("Simple Application");

    JavaSparkContext sc = new JavaSparkContext(conf);

    JavaRDD<String> logData = sc.textFile(logFile).cache();

    long numAs = logData.filter(new Function<String, Boolean>() {

      public Boolean call(String s) { return s.contains("a"); }

    }).count();

    long numBs = logData.filter(new Function<String, Boolean>() {

      public Boolean call(String s) { return s.contains("b"); }

    }).count();

    System.out.println("Lines with a: " + numAs + ", lines with b: " + numBs);

  }

}
```

Spark Self-contained applications

- **Java Spark API: configuration of Spark application**

```
String logFile = "YOUR_SPARK_HOME/README.md";

SparkConf conf = new SparkConf().setAppName("Simple Application");

JavaSparkContext sc = new JavaSparkContext(conf);

JavaRDD<String> logData = sc.textFile(logFile).cache();
```

Spark Self-contained applications

- **Java Spark API: Spark actions**

```
long numAs = logData.filter(new Function<String, Boolean>() {
  public Boolean call(String s) { return s.contains("a"); }
}).count();
long numBs = logData.filter(new Function<String, Boolean>() {
  public Boolean call(String s) { return s.contains("b"); }
}).count();
```

Spark Self-contained applications

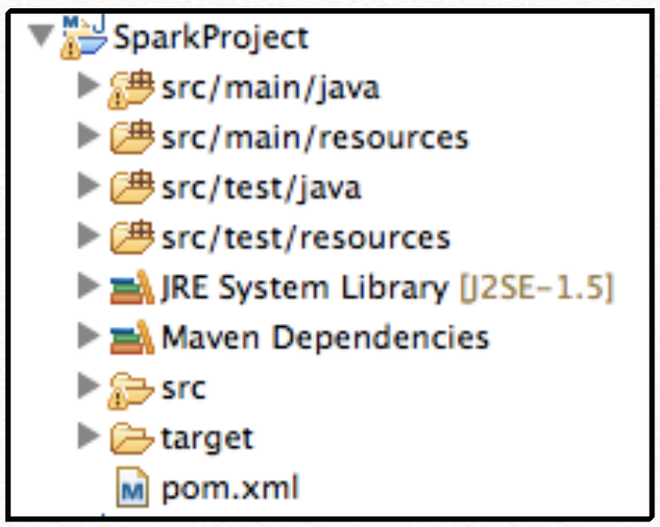
- **SimpleApp.java**

create logData: an Object like [line1, line2, line3, ...]
sopra la panca la capra campa, sotto la panca la capra crepa

Lines with **a**: 1, lines with **b**: 0

Spark Self-contained applications

- Maven Project



- pom.xml

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>edu.berkeley</groupId>
  <artifactId>simple-project</artifactId>
  <version>1.0</version>
  <dependencies>
    <dependency> <!-- Spark dependency -->
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.10</artifactId>
      <version>1.6.1</version>
    </dependency>

    <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming_2.10</artifactId>
            <version>1.6.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql_2.10</artifactId>
            <version>1.6.1</version>
        </dependency>
  </dependencies>
</project>
```

Spark Running - standalone

- Running Java Spark applications:

```
$:~spark-*/bin/spark-submit

                --class "SimpleApp"

                --master local[4]

                SparkProject-1.0.jar
```

- **output [terminal]**

```
Lines with a: 46, Lines with b: 23
```

Spark Running - standalone (PICO)

- Running Java Spark applications:

```
#!/bin/bash

#PBS -A train_bigdat16
#PBS -l walltime=00:05:00
#PBS -l select=1:ncpus=20:mem=96GB
#PBS -q parallel

$HOME/spark-1.6.1-bin-hadoop2.4/bin/spark-submit
            --class "simple.SimpleApp"
            --master local[4]
            $HOME/simple-project-1.0.jar
```

- **output [terminal]**

```
Lines with a: 46, Lines with b: 23
```

Spark Running - YARN

- Running Java Spark applications:

```
$:~spark-*/bin/spark-submit

                --class "SimpleApp"

                --master yarn

                SparkProject-1.0.jar
```

- **output [terminal]**

```
Lines with a: 46, Lines with b: 23
```

Spark Running - YARN (PICO)

- Running Java Spark applications:

```
#!/bin/bash

#PBS -A train_bigdat16
#PBS -l walltime=00:02:00
#PBS -l select=1:ncpus=20:mem=96GB
#PBS -q parallel

## Environment configuration
module load profile/advanced hadoop/2.5.1
# Configure a new HADOOP instance using PBS job information
$MYHADOOP_HOME/bin/myhadoop-configure.sh -c $HADOOP_CONF_DIR
# Start the Datanode, Namenode, and the Job Scheduler
$HADOOP_HOME/sbin/start-dfs.sh

$HADOOP_HOME/bin/hdfs dfs -mkdir /user
$HADOOP_HOME/bin/hdfs dfs -mkdir /user/rdevirgi

$HADOOP_HOME/bin/hdfs dfs -put $HADOOP_HOME/etc/hadoop input
```

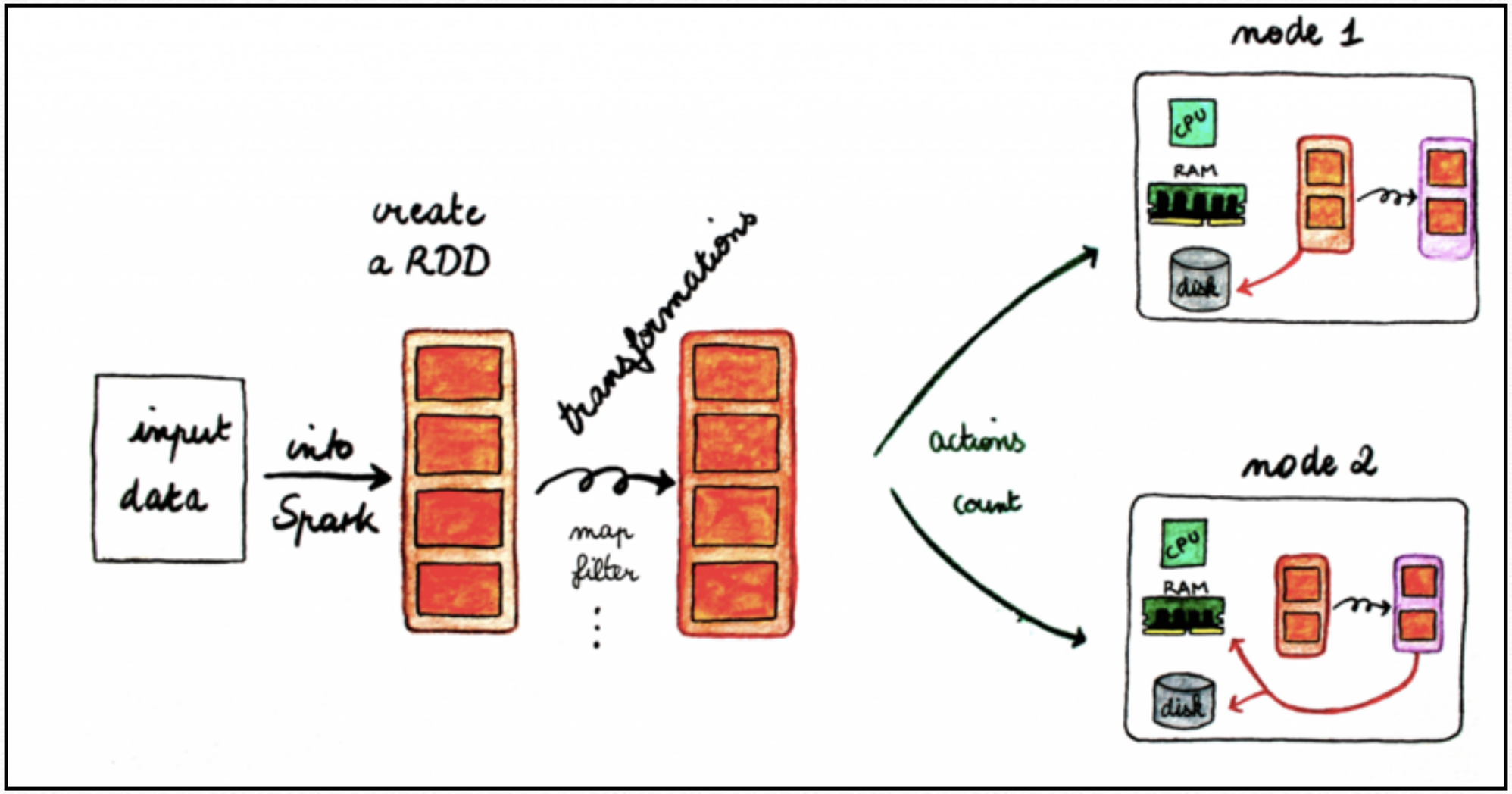
Spark Running - YARN (PICO)

- Running Java Spark applications:

```
$HADOOP_HOME/sbin/start-yarn.sh

$HADOOP_HOME/bin/hdfs dfs -mkdir input

$HADOOP_HOME/bin/hdfs dfs -put $HOME/spark-1.6.1-bin-hadoop2.4/
README.md input

$HOME/spark-1.6.1-bin-hadoop2.4/bin/spark-submit
            --class "simple.SimpleApp"
            --master yarn
            $HOME/simple-project-1.0.jar

# Stop HADOOP services
$MYHADOOP_HOME/bin/myhadoop-shutdown.sh
```

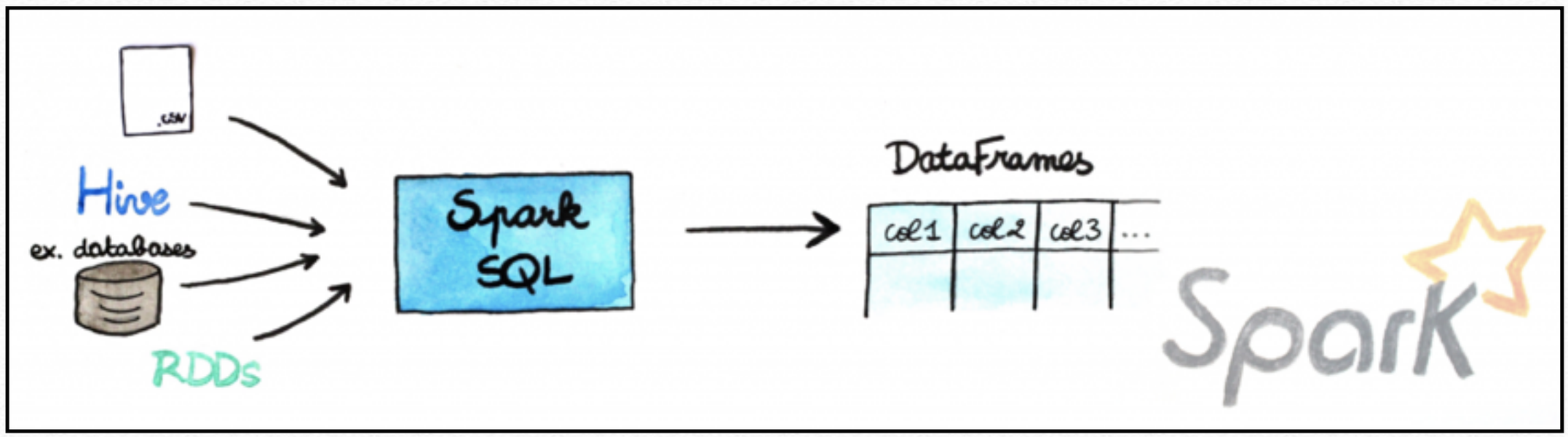
Exercises

SPARK Core API



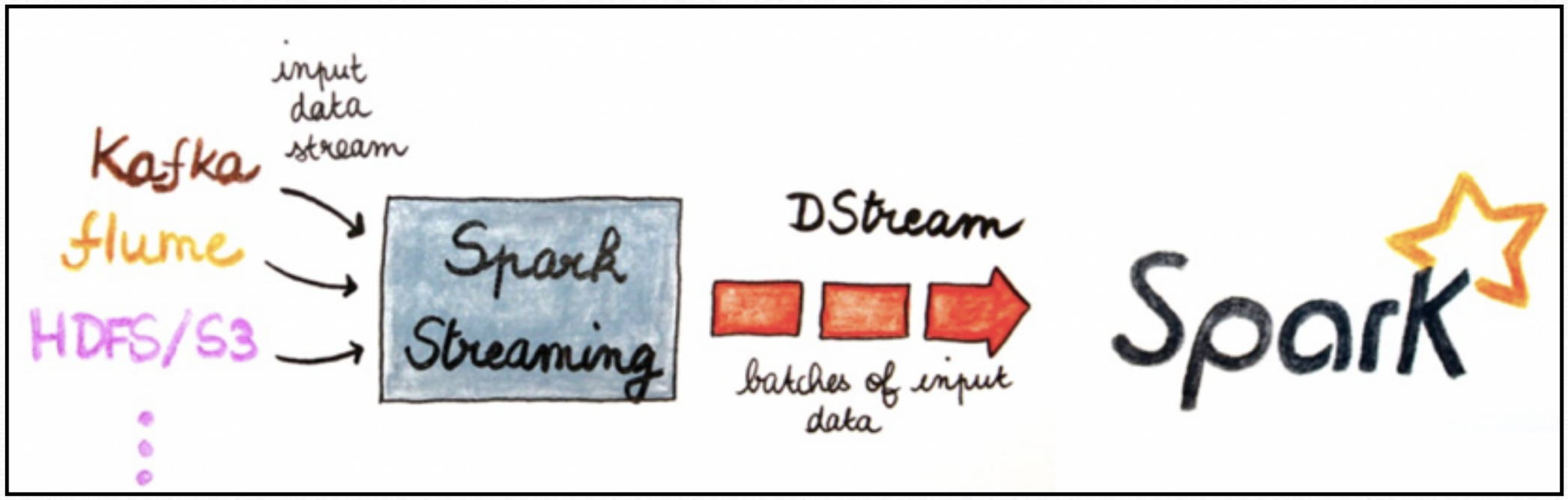
Exercises

SPARK SQL (DataFrame)



Exercises

SPARK Streaming



Exercises

SPARK Core API: Word Count

```
public final class WordCount {
  private static final Pattern SPACE = Pattern.compile(" ");

  public static void main(String[] args) throws Exception {

    if (args.length < 1) {
      System.err.println("Usage: JavaWordCount <file>");
      System.exit(1);
    }

    SparkConf sparkConf = new SparkConf().setAppName("JavaWordCount");
    JavaSparkContext ctx = new JavaSparkContext(sparkConf);
    JavaRDD<String> lines = ctx.textFile(args[0], 1);

    JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {
      @Override
      public Iterable<String> call(String s) {
        return Arrays.asList(SPACE.split(s));
      }
    });
```

Exercises

SPARK Core API: Word Count

```
JavaPairRDD<String, Integer> ones = words.mapToPair(
  new PairFunction<String, String, Integer>() {
    @Override
    public Tuple2<String, Integer> call(String s) {
      return new Tuple2<>(s, 1);
    }
  });

JavaPairRDD<String, Integer> counts = ones.reduceByKey(
  new Function2<Integer, Integer, Integer>() {
    @Override
    public Integer call(Integer i1, Integer i2) {
      return i1 + i2;
    }
  });

List<Tuple2<String, Integer>> output = counts.collect();
for (Tuple2<?,?> tuple : output) {
  System.out.println(tuple._1() + ": " + tuple._2());
}
ctx.stop();
  }
}
```

Exercises

SPARK Core API: Tweet Mining

Now we use a dataset with 8198 tweets.
Here an example of a tweet (json):


```
{"id":"572692378957430785",
 "user":"Srkian_nishu :)",
 "text":"@always_nidhi @YouTube no i dnt understand bt i
         loved of this mve is rocking",
 "place":"Orissa",
 “country":"India"
 }
```


We want to make some computations on the tweets:

- Find all the persons mentioned on tweets
- Count how many times each person is mentioned
- Find the 10 most mentioned persons by descending order

Exercises

SPARK Core API: Tweet Mining

```
public class TweetMining {

  private String pathToFile;

  public TweetMining(String file){
      this.pathToFile = file;
  }

   //  Load the data from the text file and return an RDD of Tweet

  public JavaRDD<Tweet> loadData() { }

   // Find all the persons mentioned on tweets

  public JavaRDD<String> mentionOnTweet() { }

  //  Count how many times each person is mentioned

  public JavaPairRDD<String, Integer> countMentions() { }

  //  Find the 10 most mentioned persons by descending order

  public List<Tuple2<Integer, String>> top10mentions() { }
}
```

Exercises

SPARK Core API: Tweet Mining

```
public class Tweet implements Serializable {

  long id; String user; String userName; String text;
  String place; String country; String lang;

  public String getUserName() { return userName; }

  public String getLang() { return lang; }

  public long getId() { return id; }

  public String getUser() { return user;}

  public String getText() { return text; }

  public String getPlace() { return place; }

  public String getCountry() { return country; }


  @Override
  public String toString(){
    return getId() + ", " + getUser() + ", " + getText() + ", " + getPlace() + ", " +
           getCountry();
  }
}
```

Exercises

- **SPARK Core API: Tweet Mining**

```
public class Parse {

  public static Tweet parseJsonToTweet(String jsonLine) {
    ObjectMapper objectMapper = new ObjectMapper();
    Tweet tweet = null;

    try {
      tweet = objectMapper.readValue(jsonLine, Tweet.class);
    } catch (IOException e) {
      e.printStackTrace();
    }
    return tweet;
  }

}
```

Exercises

SPARK Core API: Tweet Mining (Java 1.7 or later)

```
public JavaRDD<Tweet> loadData() {
    // create spark configuration and spark context
    SparkConf conf = new SparkConf()
                             .setAppName("Tweet mining");
                             //.setMaster("local[*]");

    JavaSparkContext sc = new JavaSparkContext(conf);

    JavaRDD<Tweet> tweets = sc.textFile(pathToFile)
                              .map(new Function<String, Tweet>() {

                            @Override
                            public Tweet call(String line) throws Exception
{
                                return Parse.parseJsonToTweet(line);
                            }
                        });

    return tweets;
  }
```

Exercises

- **SPARK Core API: Tweet Mining (LAMBDA Java 1.8)**

```
public JavaRDD<Tweet> loadData() {
    // create spark configuration and spark context
    SparkConf conf = new SparkConf()
                             .setAppName("Tweet mining");
                             //.setMaster("local[*]");

   JavaSparkContext sc = new JavaSparkContext(conf);

   JavaRDD<Tweet> tweets = sc.textFile(pathToFile)
                             .map(line -> Parse.parseJsonToTweet(line));

   return tweets;
}
```

Exercises

SPARK Core API: Tweet Mining (Java 1.7 or later)

```
public JavaRDD<String> mentionOnTweet() {
    JavaRDD<Tweet> tweets = loadData();

    JavaRDD<String> mentions = tweets.flatMap(new FlatMapFunction<Tweet,
String>() {
        @Override
        public Iterable<String> call(Tweet tweet) throws Exception {
            return Arrays.asList(tweet.getText().split(" "));
        }
    })
         .filter(new Function<String, Boolean>() {
                        @Override
                        public Boolean call(String word) throws Exception {
                        return word.startsWith("@") && word.length() > 1;
                                }
                            });

   System.out.println("mentions.count() " + mentions.count());
   return mentions;

}
```

Exercises

SPARK Core API: Tweet Mining (Java 1.8)

```
public JavaRDD<String> mentionOnTweet() {
    JavaRDD<Tweet> tweets = loadData();

     JavaRDD<String> mentions =
          tweets.flatMap(tweet -> Arrays.asList(tweet.getText()
                .split(" ")))
                .filter(word -> word.startsWith("@") && word.length() > 1);

   System.out.println("mentions.count() " + mentions.count());
   return mentions;

}
```

Exercises

- **SPARK Core API: Tweet Mining (Java 1.7 or later)**

```
public JavaPairRDD<String, Integer> countMentions() {
    JavaRDD<String> mentions = mentionOnTweet();

    JavaPairRDD<String, Integer> mentionCount = mentions.mapToPair(new
PairFunction<String, String, Integer>() {
        @Override
        public Tuple2<String, Integer> call(String mention) throws Exception {
            return new Tuple2<>(mention, 1);
        }
    })
          .reduceByKey(new Function2<Integer, Integer, Integer>() {
                @Override
                public Integer call(Integer x, Integer y) throws Exception {
                        return x + y;
                                                        }
                                                    });

    return mentionCount;
}
```

Exercises

SPARK Core API: Tweet Mining (Java 1.8)

```
public JavaPairRDD<String, Integer> countMentions() {
    JavaRDD<String> mentions = mentionOnTweet();

    JavaPairRDD<String, Integer> mentionCount =
            mentions.mapToPair(mention -> new Tuple2<>(mention, 1))
                    .reduceByKey((x, y) -> x + y);
    return mentionCount;
  }
```

Exercises

- **SPARK Core API: Tweet Mining (Java 1.7 or later)**

```
public List<Tuple2<Integer, String>> top10mentions() {
    JavaPairRDD<String, Integer> counts = countMentions();

    List<Tuple2<Integer, String>> mostMentioned =
          counts.mapToPair(new PairFunction<Tuple2<String, Integer>, Integer,
String>() {
                                    @Override
        public Tuple2<Integer, String> call(Tuple2<String, Integer> pair) throws
Exception {
            return new Tuple2<>(pair._2(), pair._1());
        }
    })
                                                          .sortByKey(false)
                                                          .take(10);

    return mostMentioned;
  }
```

Exercises

SPARK Core API: Tweet Mining (Java 1.8)

```
public List<Tuple2<Integer, String>> top10mentions() {
    JavaPairRDD<String, Integer> counts = countMentions();

    List<Tuple2<Integer, String>> mostMentioned =
          counts.mapToPair(pair -> new Tuple2<>(pair._2(), pair._1()))
                                                    .sortByKey(false)
                                                    .take(10);

    return mostMentioned;
  }
```

Exercises

Michael, 29
Andy, 30
Justin, 19

SPARK DataFrame: SparkSQL

```
public class SparkSQL {
  public static class Person implements Serializable {
    private String name;
    private int age;

    public String getName() {
      return name;
    }

    public void setName(String name) {
      this.name = name;
    }

    public int getAge() {
      return age;
    }

    public void setAge(int age) {
      this.age = age;
    }
  }
```

Exercises

SPARK DataFrame: SparkSQL

```
public static void main(String[] args) throws Exception {

        if (args.length < 2) {
            System.err.println("Usage: JavaSparkSQL <filetxt> <filejson>");
            System.exit(1);
        }

    SparkConf sparkConf = new SparkConf().setAppName("JavaSparkSQL");
    JavaSparkContext ctx = new JavaSparkContext(sparkConf);
    SQLContext sqlContext = new SQLContext(ctx);
```

```
Michael, 29
Andy, 30
Justin, 19
```

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

Exercises

Michael, 29
Andy, 30
Justin, 19

SPARK DataFrame: SparkSQL

```
System.out.println("=== Data source: RDD ===");
    // Load a text file and convert each line to a Java Bean.
    JavaRDD<Person> people = ctx.textFile(args[0]).map(
      new Function<String, Person>() {
        @Override
        public Person call(String line) {
          String[] parts = line.split(",");

          Person person = new Person();
          person.setName(parts[0]);
          person.setAge(Integer.parseInt(parts[1].trim()));

          return person;
        }
      });
```

Exercises

Michael, 29
Andy, 30
Justin, 19

SPARK DataFrame: SparkSQL

```
// Apply a schema to an RDD of Java Beans and register it as a table.
    DataFrame schemaPeople = sqlContext.createDataFrame(people, Person.class);
    schemaPeople.registerTempTable("people");

    // SQL can be run over RDDs that have been registered as tables.
    DataFrame teenagers =
        sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19");

    // The results of SQL queries are DataFrames and support all the normal RDD operations.
    // The columns of a row in the result can be accessed by ordinal.
    List<String> teenagerNames = teenagers.toJavaRDD().map(new Function<Row, String>() {
      @Override
      public String call(Row row) {
        return "Name: " + row.getString(0);
      }
    }).collect();
    for (String name: teenagerNames) {
      System.out.println(name);
    }
```

Exercises

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

SPARK DataFrame: SparkSQL

```
System.out.println("=== Data source: JSON Dataset ===");
    // A JSON dataset is pointed by path.
    // The path can be either a single text file or a directory storing text
files.
    String path = args[1];
    // Create a DataFrame from the file(s) pointed by path
    DataFrame peopleFromJsonFile = sqlContext.read().json(path);

    // Because the schema of a JSON dataset is automatically inferred, to
write queries,
    // it is better to take a look at what is the schema.
    peopleFromJsonFile.printSchema();
    // The schema of people is ...
    // root
    //  |-- age: IntegerType
    //  |-- name: StringType
```

Exercises

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

SPARK DataFrame: SparkSQL

```
    // Register this DataFrame as a table.
    peopleFromJsonFile.registerTempTable("people");

    // SQL statements can be run by using the sql methods provided by sqlContext.
    DataFrame teenagers3 =
        sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19");

    // The results of SQL queries are DataFrame and support all the normal RDD
operations.
    // The columns of a row in the result can be accessed by ordinal.
    teenagerNames = teenagers3.toJavaRDD().map(new Function<Row, String>() {
      @Override
      public String call(Row row) { return "Name: " + row.getString(0); }
    }).collect();
    for (String name: teenagerNames) {
      System.out.println(name);
    }
```

Exercises

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

SPARK DataFrame: SparkSQL

```
// Alternatively, a DataFrame can be created for a JSON dataset represented by
    // a RDD[String] storing one JSON object per string.
    List<String> jsonData = Arrays.asList(
 "{\"name\":\"Yin\",\"address\":{\"city\":\"Columbus\",\"state\":\"Ohio\"}}");
    JavaRDD<String> anotherPeopleRDD = ctx.parallelize(jsonData);
    DataFrame peopleFromJsonRDD = sqlContext.read().json(anotherPeopleRDD.rdd());

    // Take a look at the schema of this new DataFrame.
    peopleFromJsonRDD.printSchema();
    // The schema of anotherPeople is ...
    // root
    //  |-- address: StructType
    //  |    |-- city: StringType
    //  |    |-- state: StringType
    //  |-- name: StringType
```

Exercises

{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}

SPARK DataFrame: SparkSQL

```
peopleFromJsonRDD.registerTempTable("people2");

    DataFrame peopleWithCity = sqlContext.sql("SELECT name, address.city FROM people2");
    List<String> nameAndCity = peopleWithCity.toJavaRDD().map(new Function<Row, String>() {
      @Override
      public String call(Row row) {
        return "Name: " + row.getString(0) + ", City: " + row.getString(1);
      }
    }).collect();
    for (String name: nameAndCity) {
      System.out.println(name);
    }

    ctx.stop();
  }
}
```

Exercises

SPARK Streaming: SparkSQLStreaming

```
/**
 * Use DataFrames and SQL to count words in UTF8 encoded, '\n'
delimited text received from the
 * network every second.
 *
 * Usage: JavaSqlNetworkWordCount <hostname> <port>
 * <hostname> and <port> describe the TCP server that Spark
Streaming would connect to receive data.
 *
 * To run this on your local machine, you need to first run a
Netcat server
 *    `$ nc -lk 9999`
 * and then run the example
 *    `$ SparkSQLStreaming localhost 9999`
 */
```

Exercises

SPARK Streaming: SparkSQLStreaming

```
public final class SparkSQLStreaming {
  private static final Pattern SPACE = Pattern.compile(" ");

  public static void main(String[] args) {
    if (args.length < 2) {
      System.err.println("Usage: SparkSQLStreaming <hostname> <port>");
      System.exit(1);
    }

    //StreamingExamples.setStreamingLogLevels();

    // Create the context with a 1 second batch size
    SparkConf sparkConf = new SparkConf().setAppName("SparkSQLStreaming");
    JavaStreamingContext ssc =
                new JavaStreamingContext(sparkConf, Durations.seconds(1));
```

Exercises

SPARK Streaming: SparkSQLStreaming

```
// Create a JavaReceiverInputDStream on target ip:port and count the
    // words in input stream of \n delimited text (eg. generated by 'nc')
    // Note that no duplication in storage level only for running locally.
    // Replication necessary in distributed scenario for fault tolerance.
    JavaReceiverInputDStream<String> lines =
                ssc.socketTextStream(args[0],
                                     Integer.parseInt(args[1]),
                                     StorageLevels.MEMORY_AND_DISK_SER);
  JavaDStream<String> words =
    lines.flatMap(new FlatMapFunction<String, String>() {
    @Override
    public Iterable<String> call(String x) {
      return Arrays.asList(SPACE.split(x));
    }
  });
```

Exercises

SPARK Streaming: SparkSQLStreaming

```
// Convert RDDs of the words DStream to DataFrame and run SQL query
    words.foreachRDD(new VoidFunction2<JavaRDD<String>, Time>() {
      @Override
      public void call(JavaRDD<String> rdd, Time time) {
        SQLContext sqlContext = JavaSQLContextSingleton.getInstance(rdd.context());

        // Convert JavaRDD[String] to JavaRDD[bean class] to DataFrame
        JavaRDD<JavaRecord> rowRDD = rdd.map(new Function<String, JavaRecord>() {
          @Override
          public JavaRecord call(String word) {
            JavaRecord record = new JavaRecord();
            record.setWord(word);
            return record;
          }
        });
```

Exercises

SPARK Streaming: SparkSQLStreaming

```
DataFrame wordsDataFrame = sqlContext.createDataFrame(rowRDD, JavaRecord.class);

        // Register as table
        wordsDataFrame.registerTempTable("words");

        // Do word count on table using SQL and print it
        DataFrame wordCountsDataFrame =
            sqlContext.sql("select word, count(*) as total from words group by word");
        System.out.println("========= " + time + "=========");
        wordCountsDataFrame.show();
      }
    });

    ssc.start();
    ssc.awaitTermination();
  }
}
```

Exercises

SPARK Streaming: SparkSQLStreaming

```
/** Java Bean class to be used with the example JavaSqlNetworkWordCount. */
public class JavaRecord implements java.io.Serializable {
  private String word;

  public String getWord() {
    return word;
  }

  public void setWord(String word) {
    this.word = word;
  }
}
```

High-Speed In-Memory Analytics over Hadoop and Hive Data

20/04/2016 - Big Data 2016