TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

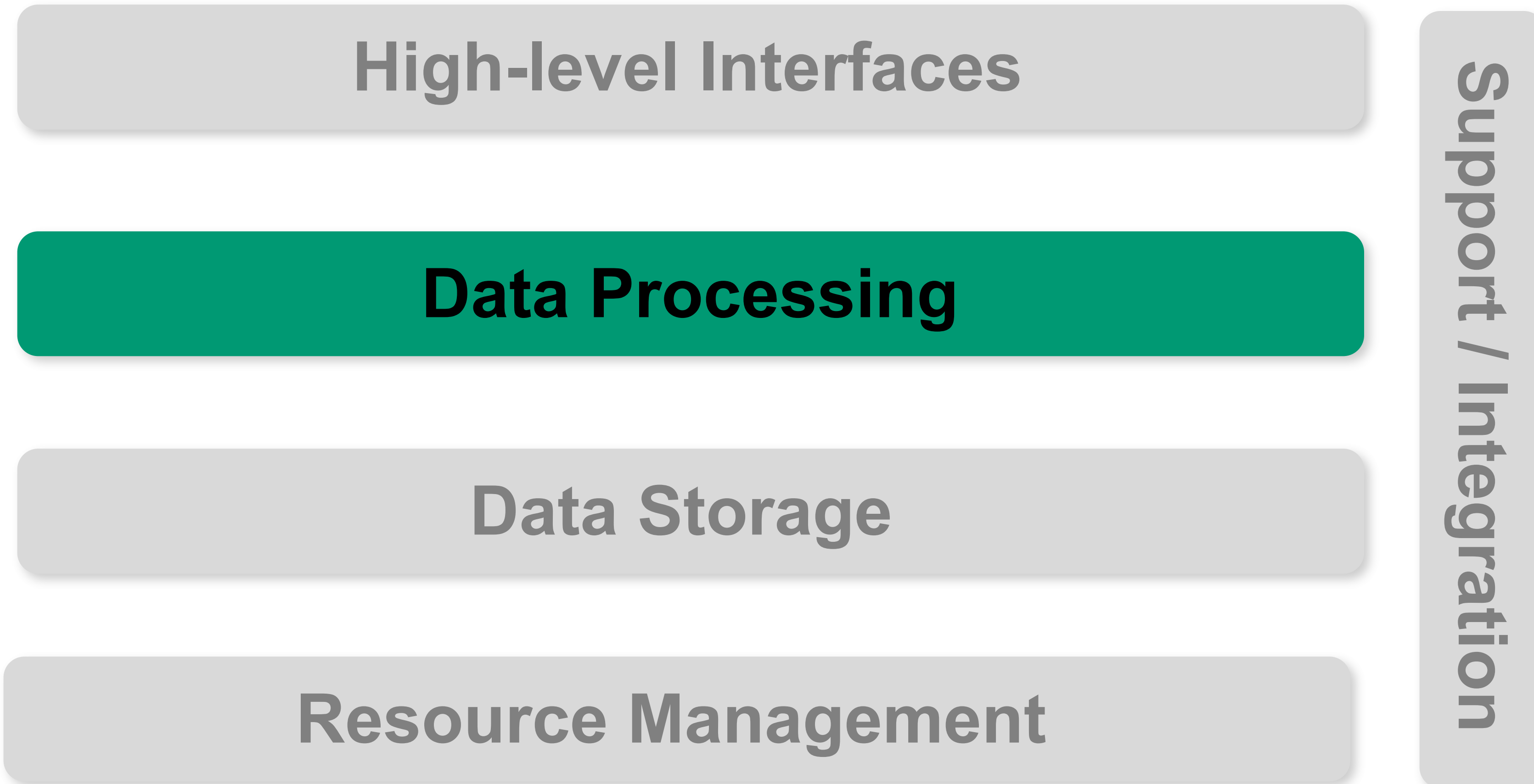
Apache Spark: Hands-on Session

A.A. 2023/24

Matteo Nardelli

Laurea Magistrale in Ingegneria Informatica - II anno

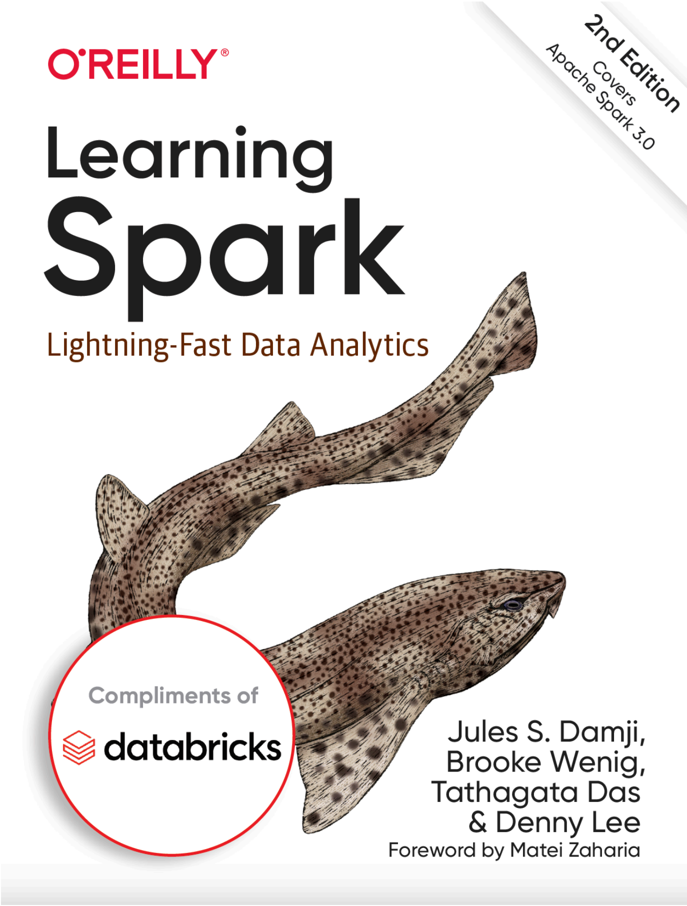
The reference Big Data stack



Main reference for this lecture

J.S. Damji, B. Wenig, T. Das, D. Lee,
"Learning Spark: Lightning-fast Data Analytics"
2nd ed., O'Reilly Media, 2020.

H.Karau, A. Konwinski, P. Wendell,
M. Zaharia, "Learning Spark"
O'Reilly Media, 2015.



Java 8: Lambda Expressions

- You're usually trying to pass functionality as an argument to another method
  - e.g., what action should be taken when someone clicks a button
- **Lambda expressions** enable to treat functionality as method argument, or code as data

Java 8: Lambda Expressions

Example: a social networking application.

- You want to create a feature that enables an administrator to perform any kind of action, such as sending a message, on members of the social networking application that satisfy certain criteria

- Suppose that members of this social networking application are represented by the following Person class:

```
public class Person {
 public enum Sex { MALE, FEMALE }

 String name;
 LocalDate birthday;
 Sex gender;
 String emailAddress;

 public int getAge() { ... }
 public void printPerson() { ... }

}
```

Java 8: Lambda Expressions

- Suppose that the members of your social networking application are stored in a List instance

Approach 1: Create methods that search for members matching one characteristic

```
public static void invitePersons(List<Person> roster,
                                                        int age){
    for (Person p : roster) {
        if (p.getAge() >= age) {
                p.sendMessage();
        }
    }
}
```

Java 8: Lambda Expressions

Approach 2: Decouple the matching criteria and specify them in a local class

```
public static void invitePersons(List<Person> roster,
                                 CheckPerson tester){
    for (Person p : roster) {
        if (tester.test(p)) { p.sendMessage(); }
    }
}

interface CheckPerson {
    boolean test(Person p);
}

class CheckEligiblePerson implements CheckPerson {
    public boolean test(Person p) {
        return p.getAge() >= 18 && p.getAge() <= 25;
    }
}
```

Java 8: Lambda Expressions

Approach 3: Specify search criteria code in an anonymous class

```
invitePersons(
 roster,
 new CheckPerson() {
        public boolean test(Person p) {
            return p.getAge() >= 18 && p.getAge() <= 25;
        }
    }
);
```

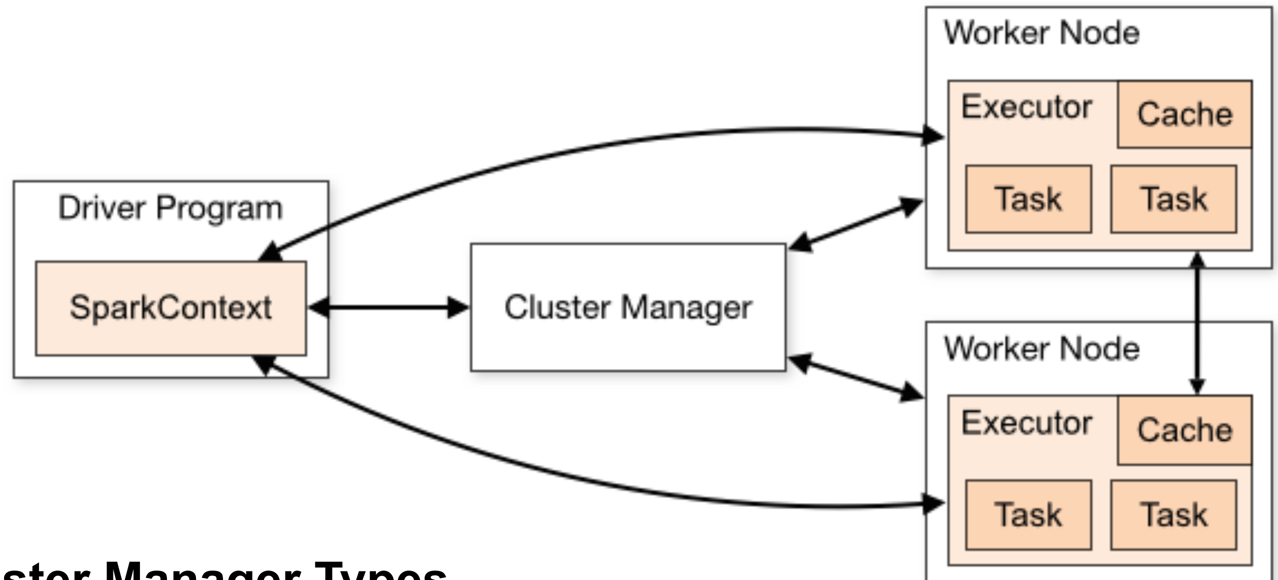
Approach 4: Specify search criteria code with a lambda expression

```
invitePersons(
 roster,
 (Person p) -> p.getAge() >= 18 && p.getAge() <= 25
);
```

Apache Spark

Spark Cluster

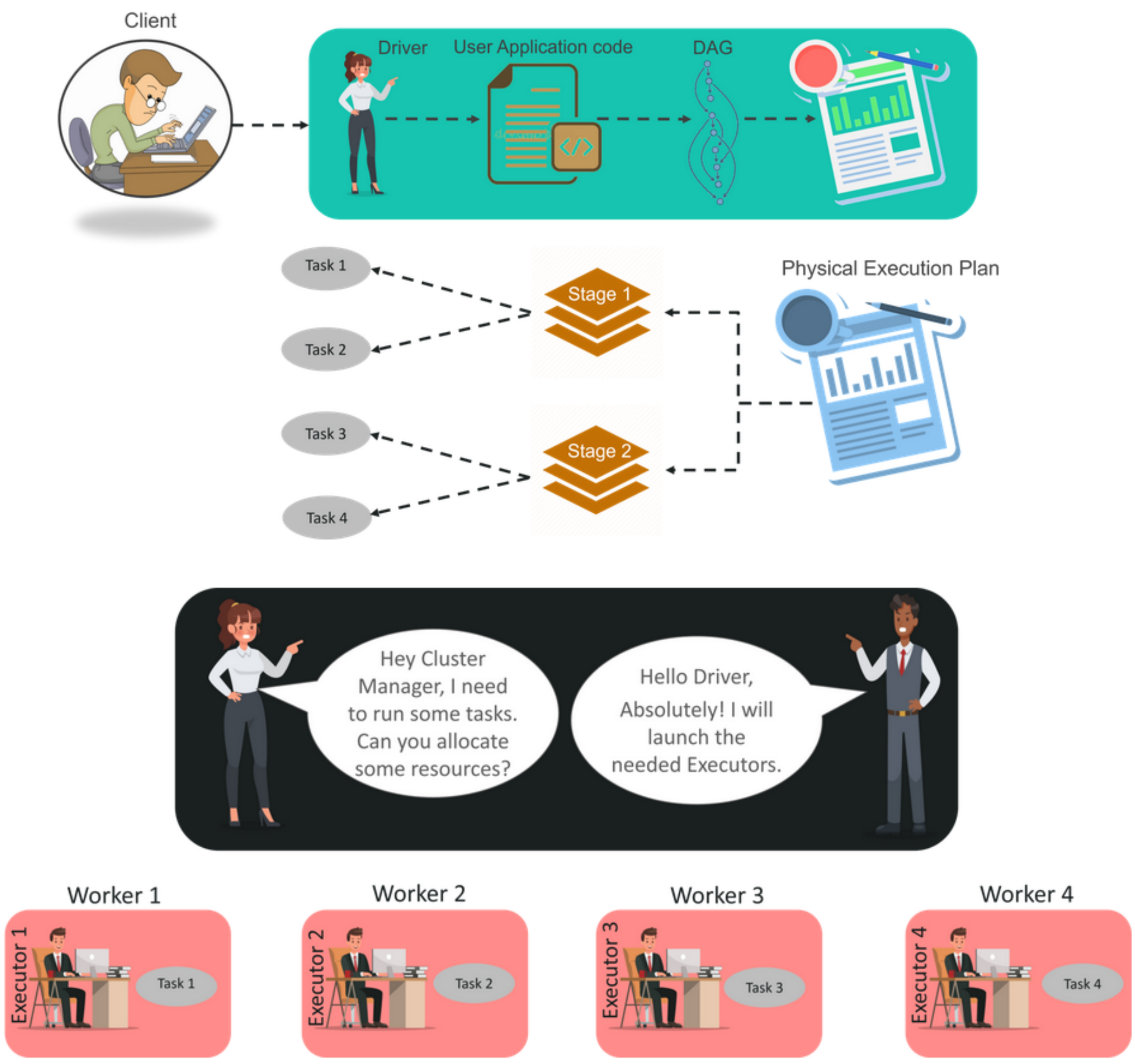
- Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in a Spark program (called the *driver program*).



Cluster Manager Types

- Standalone: a simple cluster manager included with Spark
- Apache Mesos
- Hadoop YARN

Spark Cluster



Resilient Distributed Dataset (RDD)

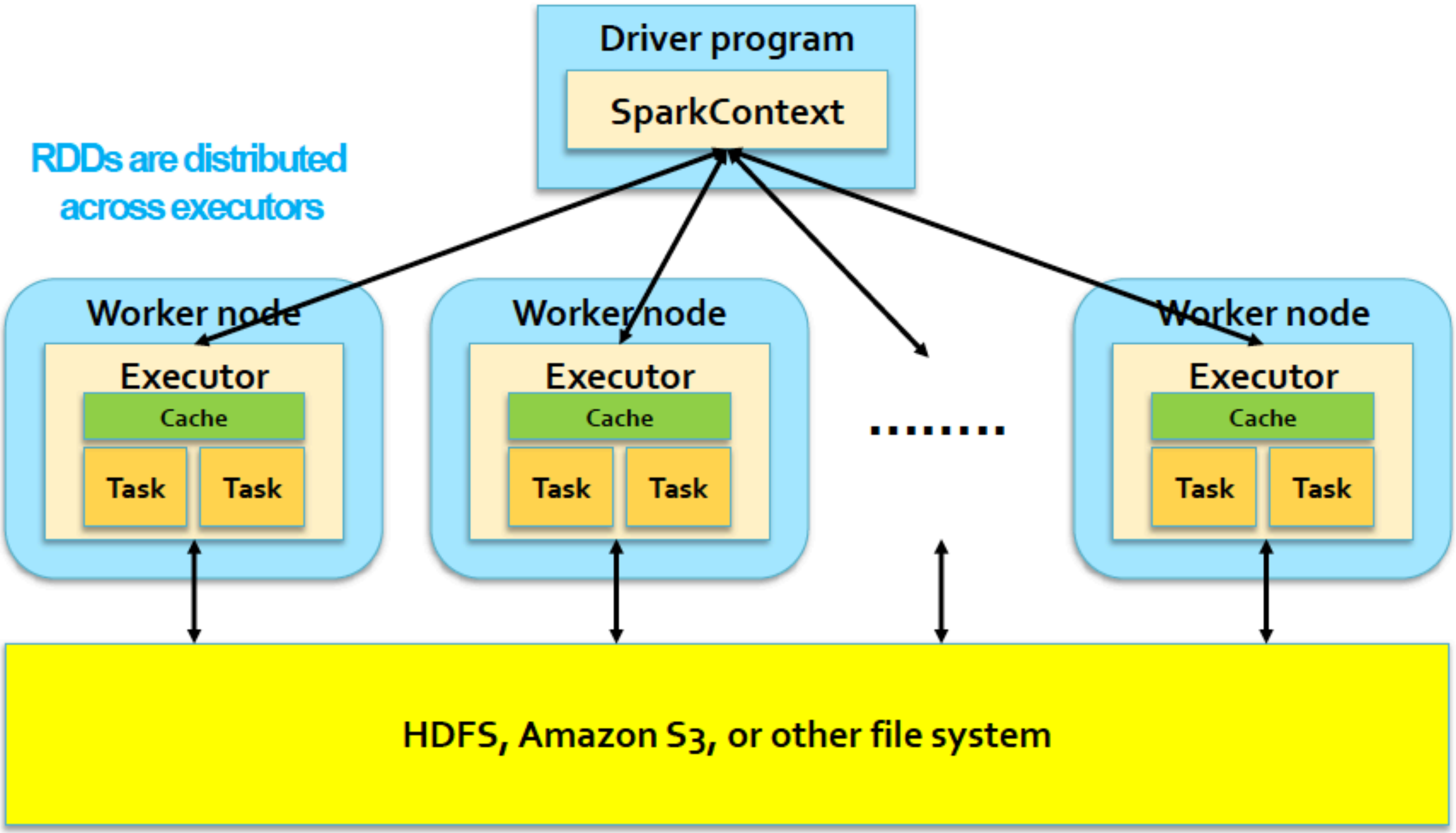
- The primary abstraction in Spark: a distributed memory abstraction

- Immutable, partitioned collection of elements
  - Like a LinkedList <MyObjects>
  - Operated on in parallel
  - Cached in memory across the cluster nodes
    - Each node of the cluster that is used to run an application contains at least one partition of the RDD(s) that is (are) defined in the application

Spark and RDDs

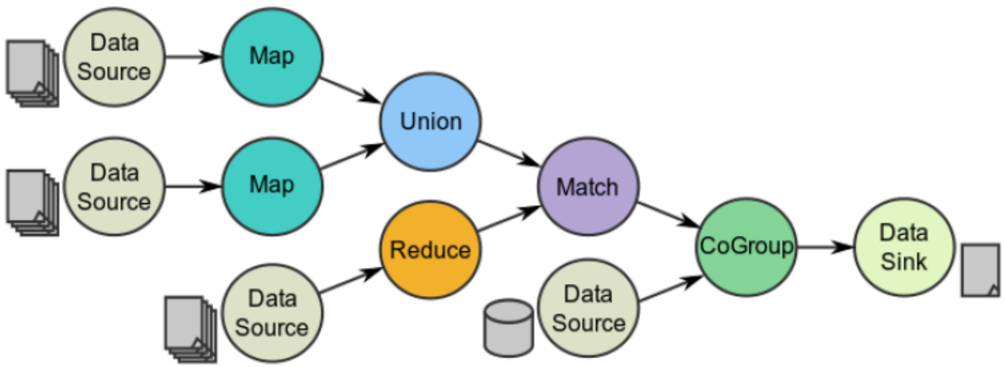
- Spark manages scheduling and synchronization of the jobs
- Manages the split of RDDs in partitions and allocates RDDs' partitions in the nodes of the cluster
- Hides complexities of fault-tolerance and slow machines
- RDDs are automatically rebuilt in case of machine failure

Spark and RDDs



Spark programming model

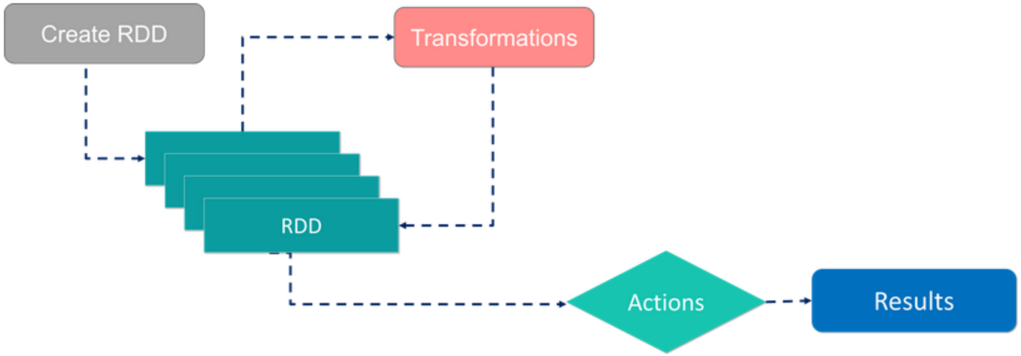
- Spark programming model is based on parallelizable operators
- Parallelizable operators are higher-order functions that execute user-defined functions in parallel
- A data flow is composed of any number of data sources, operators, and data sinks by connecting their inputs and outputs
- Job description based on DAG



Resilient Distributed Dataset (RDD)

- Spark programs are written in terms of operations on RDDs

- RDDs built and manipulated through:
  - Coarse-grained **transformations**
    - Map, filter, join, …
  - **Actions**
    - Count, collect, save, …



Spark Cluster

- You can start a standalone master server by executing:

```
$ $SPARK_HOME/sbin/start-master.sh
```

(on master node)

- Similarly, you can start one or more workers and connect them to the master via:

```
$ $SPARK_HOME/sbin/start-slave.sh <master-spark-URL>
```

(on slave nodes)

- It is also possible to start slaves from the master node:

```
# Starts a slave instance on each machine specified
#   in the conf/slaves file on the master node

$ $SPARK_HOME/sbin/start-slaves.sh
```

(on master node)

- Spark has a WebUI reachable at http://localhost:8080

Spark Cluster

- You can stop the master server by executing:

```
$ $SPARK_HOME/sbin/stop-master.sh
```

(on master node)

- Similarly, you can stop a worker via:

```
$ $SPARK_HOME/sbin/stop-slave.sh
```

(on slave nodes)

- It is also possible to stop slaves from the master node:

```
# Starts a slave instance on each machine specified
#   in the conf/slaves file on the master node

$ $SPARK_HOME/sbin/stop-slaves.sh
```

(on master node)

SparkContext using Java

The first thing a Spark program must do is to create a JavaSparkContext object, which tells Spark how to access a cluster. To create a SparkContext you first need to build a SparkConf object that contains information about your application (i.e., the appName parameter and the cluster master URL).

```
import org.apache.spark.api.java.JavaSparkContext
import org.apache.spark.api.java.JavaRDD
import org.apache.spark.SparkConf

public class SparkApp {
    public static void main(String[] args){
        SparkConf conf = new SparkConf()
                        .setMaster(master)
                                        .setAppName("appName");
        JavaSparkContext sc = new JavaSparkContext(conf);
      ...
        sc.stop();
    }
}
```

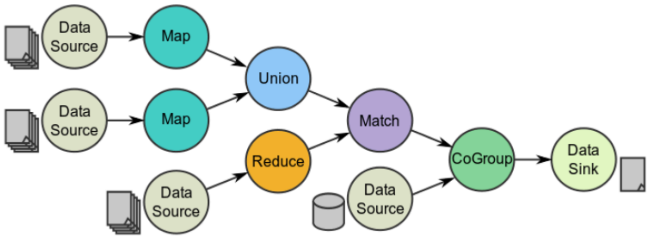
Spark Master URLs

Master URL	Meaning
`local`	Run Spark locally with one worker thread (i.e. no parallelism at all).
`local[K]`	Run Spark locally with K worker threads (ideally, set this to the number of cores on your machine).
`local[K,F]`	Run Spark locally with K worker threads and F maxFailures (see spark.task.maxFailures for an explanation of this variable)
`local[*]`	Run Spark locally with as many worker threads as logical cores on your machine.
`local[*,F]`	Run Spark locally with as many worker threads as logical cores on your machine and F maxFailures.
`spark://HOST:PORT`	Connect to the given Spark standalone cluster master. The port must be whichever one your master is configured to use, which is 7077 by default.
`spark://HOST1:PORT1,HOST2:PORT2`	Connect to the given Spark standalone cluster with standby masters with Zookeeper. The list must have all the master hosts in the high availability cluster set up with Zookeeper. The port must be whichever each master is configured to use, which is 7077 by default.
`mesos://HOST:PORT`	Connect to the given Mesos cluster. The port must be whichever one your is configured to use, which is 5050 by default. Or, for a Mesos cluster using ZooKeeper, use `mesos://zk://....` To submit with `--deploy-mode cluster`, the HOST:PORT should be configured to connect to the MesosClusterDispatcher.
`yarn`	Connect to a YARN cluster in `client` or `cluster` mode depending on the value of `--deploy-mode`. The cluster location will be found based on the `HADOOP_CONF_DIR` or `YARN_CONF_DIR` variable.
`k8s://HOST:PORT`	Connect to a Kubernetes cluster in `cluster` mode. Client mode is currently unsupported and will be supported in future releases. The `HOST` and `PORT` refer to the Kubernetes API Server. It connects using TLS by default. In order to force it to use an unsecured connection, you can use `k8s://http://HOST:PORT`.

source: https://spark.apache.org/docs/latest/submitting-applications.html

Passing Functions to Spark using Java

- Spark's API relies heavily on passing functions in the driver program to run on the cluster.
- In Java, functions are represented by classes implementing the interfaces in the org.apache.spark.api.java.function package.
- There are two ways to create such functions:
  - Implement the Function interfaces in your own class, either as an anonymous inner class or a named one, and pass an instance of it to Spark.
  - In Java 8, use lambda expressions to concisely define an implementation.



Spark: Launching Applications

```
$ ./bin/spark-submit \
        --class <main-class> \
        --master <master-url> \
        [--conf <key>=<value>] \
        <application-jar> \
        [application-arguments]
```

--class: The entry point for your application (e.g. package.WordCount)

--master: The master URL for the cluster
e.g., "local", "spark://HOST:PORT", "mesos://HOST:PORT"

--conf: Arbitrary Spark configuration property

application-jar: Path to a bundled jar including your application and all dependencies.

application-arguments: Arguments passed to the main method of your main class, if any

How to create RDDs

- RDD can be created by:
  - Parallelizing existing collections of the hosting programming language (e.g., collections and lists of Scala, Java, Python, or R)
    - Number of partitions specified by user
    - API: `parallelize`
  - From (large) files stored in HDFS or any other file system
    - One partition per HDFS block
    - API: `textFile`
  - By transforming an existing RDD
    - Number of partitions depends on transformation type
    - API: transformation operations (`map`, `filter`, `flatMap`)

How to create RDDs

- parallelize: Turn a collection into an RDD

```
val a = sc.parallelize(Array(1, 2, 3))
```

- textFile: Load text file from local file system, HDFS, or S3

```
val a = sc.textFile("file.txt")
val b = sc.textFile("directory/*.txt")
val c = sc.textFile("hdfs://namenode:9000/path/file")
```

Operations over RDD

Transformations

- Create a new dataset from and existing one.
- Lazy in nature. They are executed only when some action is performed.
- Example: map(), filter(), distinct()

Actions

- Returns to the driver program a value or exports data to a storage system after performing a computation.
- Example: count(), reduce(), collect()

Persistence

- For caching datasets in-memory for future operations. Option to store on disk or RAM or mixed.
- Functions: persist(), cache()

Operations over RDD: Transformations

Function name	Purpose	Example	Result
`map()`	Apply a function to each element in the RDD and return an RDD of the result.	`rdd.map(x => x + 1)`	`{2, 3, 4, 4}`
`flatMap()`	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	`rdd.flatMap(x => x.to(3))`	`{1, 2, 3, 2, 3, 3, 3}`
`filter()`	Return an RDD consisting of only elements that pass the condition passed to `filter()`.	`rdd.filter(x => x != 1)`	`{2, 3, 3}`
`distinct()`	Remove duplicates.	`rdd.distinct()`	`{1, 2, 3}`
`sample(withReplacement, fraction, [seed])`	Sample an RDD, with or without replacement.	`rdd.sample(false, 0.5)`	Nondeterministic

Operations over RDD: Transformations

Function name	Purpose	Example	Result
`union()`	Produce an RDD containing elements from both RDDs.	`rdd.union(other)`	`{1, 2, 3, 3, 4, 5}`
`intersection()`	RDD containing only elements found in both RDDs.	`rdd.intersection(other)`	`{3}`
`subtract()`	Remove the contents of one RDD (e.g., remove training data).	`rdd.subtract(other)`	`{1, 2}`
`cartesian()`	Cartesian product with the other RDD.	`rdd.cartesian(other)`	`{(1, 3), (1, 4), … (3,5)}`

Operations over RDD: Actions

- Actions are synchronous
- They trigger execution of RDD transformations to return values
- Until no action is fired, the data to be processed is not even accessed
- Only actions can materialize the entire process with real data
- Cause data to be returned to driver or saved to output

Table 3-4. Basic actions on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`collect()`	Return all elements from the RDD.	`rdd.collect()`	`{1, 2, 3, 3}`
`count()`	Number of elements in the RDD.	`rdd.count()`	4
`countByValue()`	Number of times each element occurs in the RDD.	`rdd.countByValue()`	`{(1, 1), (2, 1), (3, 2)}`

Operations over RDD: Actions

Function name	Purpose	Example	Result
`take(num)`	Return `num` elements from the RDD.	`rdd.take(2)`	`{1, 2}`
`top(num)`	Return the top `num` elements the RDD.	`rdd.top(2)`	`{3, 3}`
`takeOrdered(num)(ordering)`	Return `num` elements based on provided ordering.	`rdd.takeOrdered(2)(myOrdering)`	`{3, 3}`
`takeSample(withReplacement, num, [seed])`	Return `num` elements at random.	`rdd.takeSample(false, 1)`	Nondeterministic
`reduce(func)`	Combine the elements of the RDD together in parallel (e.g., `sum`).	`rdd.reduce((x, y) => x + y)`	9

Basic RDD actions

- collect: returns all the elements of the RDD as an array

```
val nums = sc.parallelize(Array(1, 2, 3))
nums.collect() // Array(1, 2, 3)
```

- take: returns an array with the first *n* elements in the RDD

```
nums.take(2) // Array(1, 2)
```

- count: returns the number of elements in the RDD

```
nums.count() // 3
```

Basic RDD actions

- reduce: aggregates the elements in the RDD using the specified function

```
nums.reduce((x, y) => x + y)
or
nums.reduce(_ + _) // 6
```

- saveAsTextFile: writes the elements of the RDD as a text file either to the local file system or HDFS

```
nums.saveAsTextFile("hdfs://file.txt")
```

Operations over RDD: Persistence

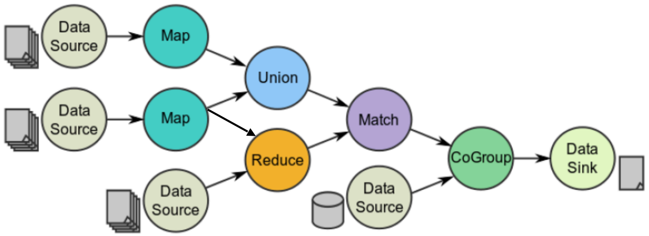
Spark RDDs are lazily evaluated, and sometimes we may wish to use the **same RDD multiple** times. Spark will recompute the RDD each time we call an action on it. This can be expensive

To avoid computing an RDD multiple times, we can ask Spark to **persist the data**. Caching is the key tool for iterative algorithms.

persist()
can specify the Storage Level for persisting an RDD;
some of the Storage Levels are:

- MEMORY_ONLY, MEMORY_AND_DISK, DISK_ONLY

cache()
is just a shortcut for persist(StorageLevel.MEMORY_ONLY)



Transformations

map()

- The map() transformation takes in a function and applies it to each element in the RDD with the result of the function being **the new value of each element** in the resulting RDD.
- We can use map() to do any number of things, from fetching the website associated with each URL in our collection to just squaring the numbers.

filter()

- The filter() transformation takes in a function and returns an RDD that only has elements that pass the filter() function.
- Makes easy to implement the *filter pattern* in MapReduce

Example: Square even numbers

Example: Square Even Numbers

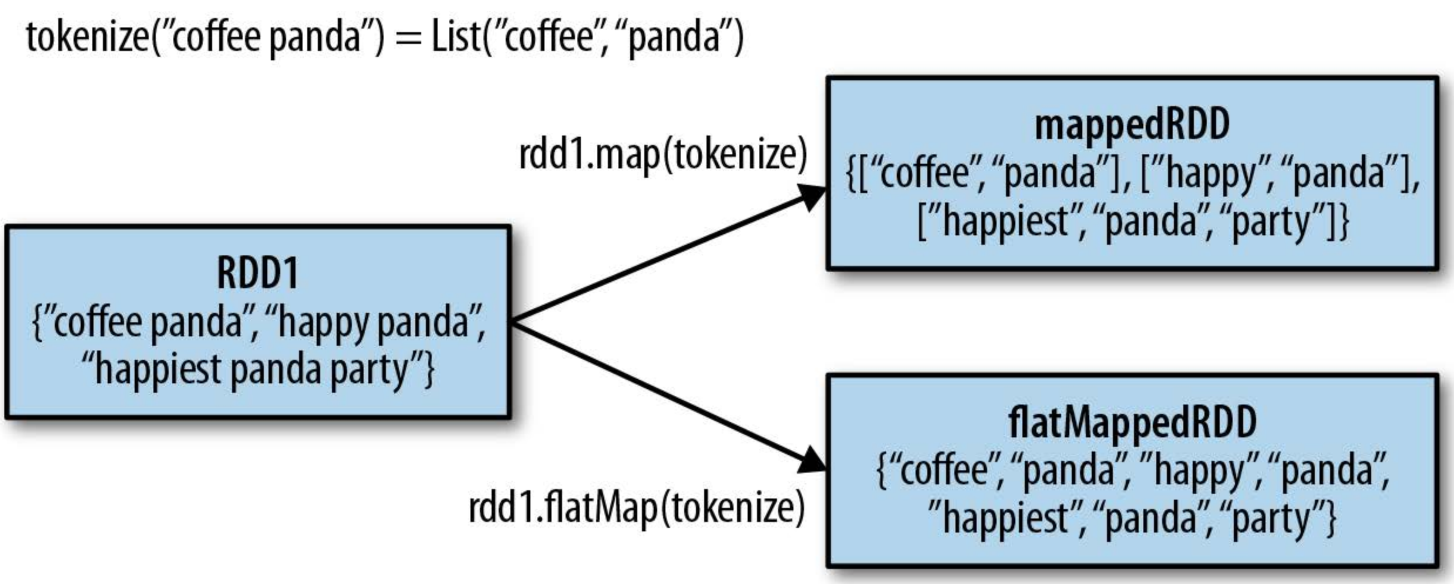
```
public class SquareEvenNumbers {
    public static void main(String[] args){
        SparkConf conf = new SparkConf()
                                .setAppName("Square Even Number");
        JavaSparkContext sc = new JavaSparkContext(conf);

        JavaRDD<Integer> input =
                sc.parallelize(Arrays.asList(1, 2, 3, 4, 5, 6));

        JavaRDD<Integer> evenNumbers =
                                input.filter(x -> (x % 2 == 0));

        JavaRDD<Integer> squaredEvenNumbers =
                                evenNumbers.map(x -> x * x);

        for (Integer i : squaredEvenNumbers.collect())
                System.out.println(i);
        sc.stop();
    }
}
```

This is only an excerpt

Transformations

flatMap()

Sometimes we want to produce **multiple output elements** for each input element. The operation to do this is called flatMap().



Actions

reduce()

This action takes a function that operates on two elements of the type in your RDD and returns **a new element of the same type**.

The function should be associative so that it can be computed in parallel.

$$a + (b + c) = (a + b) + c$$

Useful to sum, multiply, count, and aggregate the elements of a RDD.

Example (in Python): Sum all elements

```
lines = # Dstream with numbers
nums = lines.map(lambda x : int(x))
sum_nums = nums.reduce(lambda x, y: x + y)
```

Actions

reduceByKey()

When called on (K, V) pairs, return a new RDD of (K, V) pairs, where the values for each key are aggregated using the given reduce function

- Observe that, when implementing the function, we do not have to care about the key

Example: Word Count

```
public class WordCount {
      private static final Pattern SPACE = Pattern.compile(" ");
      public static void main(String[] args){

            SparkConf conf = [...]
            JavaSparkContext sc = new JavaSparkContext(conf);

            JavaRDD<String> input = [...]
```


This is only an excerpt

Example: Word Count

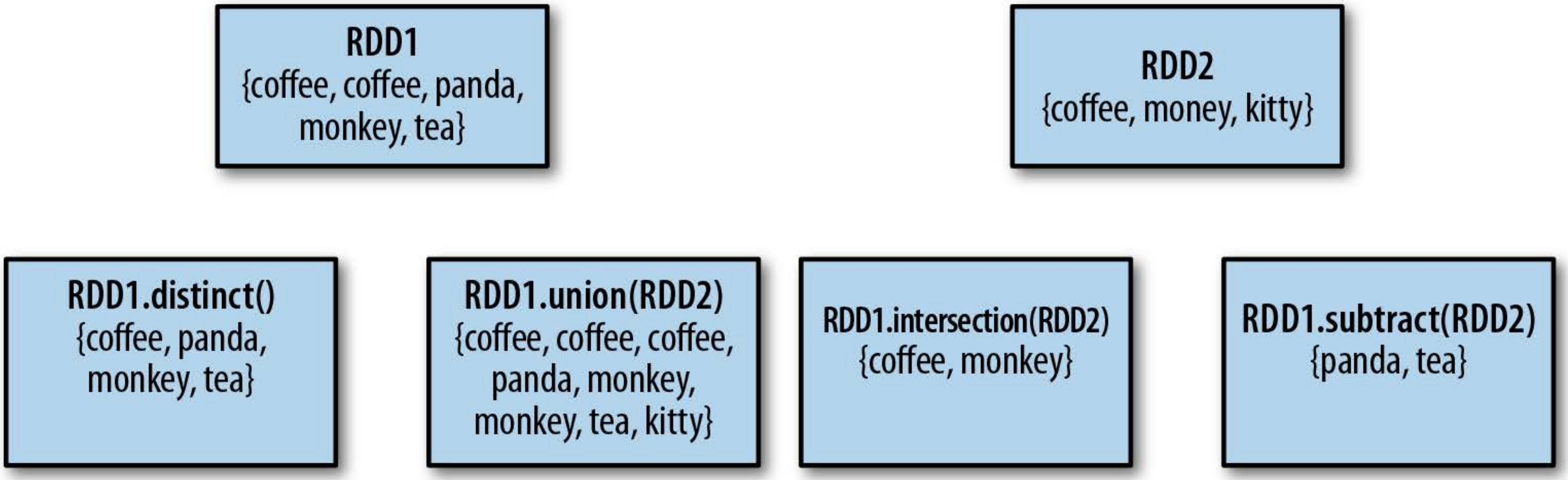
```
        // We create a RDD of words by splitting a line of text
        JavaRDD<String> words =
                input.flatMap(line ->
                  Arrays.asList(SPACE.split(line)).iterator());

        // We create the pair word, 1 to count elements using
        // the number summarization pattern
        JavaPairRDD<String, Integer> pairs =
                words.mapToPair(word -> new Tuple2<>(word, 1));

        // We reduce the elements by key (i.e., word) and count
        JavaPairRDD<String, Integer> counts =
                pairs.reduceByKey((x, y) -> x+y);

        counts.saveAsTextFile(outputPath);
        sc.stop();
    }
}
```

This is only an excerpt

Transformations

Pseudoset operations

RDDs support many of the operations of mathematical sets, such as union and intersection, even when the RDDs themselves are not properly sets



Transformations

Sample()

extracts a subset of the RDD, using two parameter: sampling with replacement, and sampling probability.

A recall from statistics: Sampling with Replacement

- Suppose we have a bowl of 100 unique numbers from 0 to 99.
- We want to select a random sample of numbers from the bowl. After we pick a number from the bowl, we can put the number aside or we can put it back into the bowl.
  - If we put the number back in the bowl, it may be selected more than once;
  - if we put it aside, it can selected only one time.
- When a population element can be selected more than one time, we are **sampling with replacement**.
- When a population element can be selected only one time, we are **sampling without replacement**.

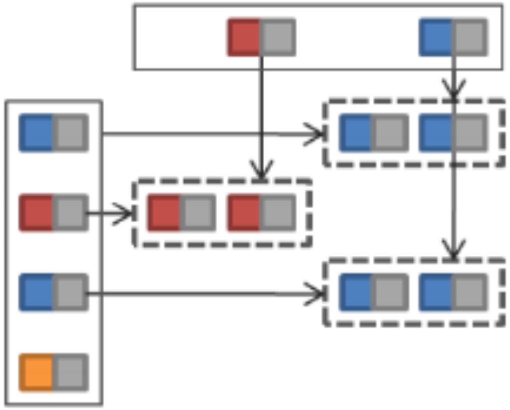
Example: DistinctAndSample

```
public class DistinctAndSample {
	[...]
	public static void main(String[] args){
		[...]

		JavaRDD<Integer> input = [...]
		JavaRDD<Integer> distinctNumbers = input.distinct();
List<Integer> distinct = distinctNumbers.collect();
		JavaRDD<Integer> sampleNumbers =
						input.sample(SAMPLING_REPLACEMENT,
								SAMPLING_PROBABILITY);
		List<Integer> sampled = sampleNumbers.collect();

		[...]
	}
}
```

This is only an excerpt

RDD transformations: join

- join Performs an equi-join on the key of two RDDs
- Join candidates are independently processed



```
val visits = sc.parallelize(Seq(("index.html", "1.2.3.4"),
                                ("about.html", "3.4.5.6"),
                                ("index.html", "1.3.3.1")))

val pageNames = sc.parallelize(Seq(("index.html", "Home"),
                                   ("about.html", "About")))

visits.join(pageNames)
  // ("index.html", ("1.2.3.4", "Home"))
  // ("index.html", ("1.3.3.1", "Home"))
  // ("about.html", ("3.4.5.6", "About"))
```

Example: SimpleJoin

```
public class SimpleJoin{
	[...]

	JavaRDD<String> transactionInputFile =
								sc.textFile(fileTransactions);

	JavaPairRDD<String, Integer> transactionPairs =
						transactionInputFile.mapToPair( [...] );

	JavaRDD<String> customerInputFile = sc.textFile(fileUsers);
JavaPairRDD<String, String> customerPairs =
						customerInputFile.mapToPair( [...] );

	List<Tuple2<String, Tuple2<Integer, String>>> result =
				transactionPairs.join(customerPairs).collect();
	[...]
}
```

This is only an excerpt

Example: Tweet Mining

Given a set of tweets, we are interested in solving two queries.

Example of tweet:

{"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"}

Query1: count the number of mentions

Query2: find the top 10 mentioned people

Example: Tweet Mining (1/3)

```
public class TweetMining {
    private static String pathToFile = "tweets.json";
    private static Pattern SPACE = Pattern.compile(" ");

    public static void main(String[] args){
        SparkConf conf = new SparkConf().setMaster("local")
                                        .setAppName("Tweet mining");
        JavaSparkContext sc = new JavaSparkContext(conf);

        JavaRDD<String> rawTweets = sc.textFile(pathToFile);
        JavaRDD<Tweet> tweets =
            rawTweets.map(line -> TweetParser.parseJson(line));
        JavaRDD<String> words =
                tweets.flatMap(tweet ->
                    Arrays.asList(SPACE.split(tweet.getText()))
                                        .iterator());
```

Example: Tweet Mining (2/3)

```
JavaRDD<String> mentions =
        words.filter(word ->
            word.startsWith("@") && word.length() > 2);

System.out.println("Query 1 - Count Mentions:"
                + mentions.distinct().count());


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

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

Example: Tweet Mining (3/3)

```
        System.out.println("Query 2 - Top 10 mentioned users");

        for (Tuple2<Integer, String> mm : mostMentioned){
                System.out.println(mm._2() + ": " + mm._1());
        }
        sc.stop();
    }
}
```

Example: Inverted Index (1/2)

We want to create an index that connects a hashtag with all users that tweeted that hashtag.

Hint: recall that in MapReduce we can obtain "for free" all elements related to the same key.

```
public class TweetMining {
      [...]
      JavaRDD<String> rawTweets = sc.textFile(pathToFile);
JavaRDD<Tweet> tweets =
            rawTweets.map(line -> TweetParser.parseJson(line));

      // For each tweet t, we extract all the hashtags
      //  and create a pair (hashtag,user)
      JavaPairRDD<String, String> pairs =
            tweets.flatMapToPair(new HashtagToTweetExtractor());
```

This is only an excerpt

Example: Inverted Index (2/2)

```
// We use the groupBy to group users by hashtag        JavaPairRDD<String,
Iterable<String>> tweetsByHashtag =
                                                        pairs.groupByKey();

    // Then return a map using the collectAsMap
    Map<String, Iterable<String>> map =
                                         tweetsByHashtag.collectAsMap();

    for(String hashtag : map.keySet()){
        System.out.println(hashtag + " -> " + map.get(hashtag));
    }

    sc.stop();
    [...]
}
```

This is only an excerpt

Example: LogAnalyzer (1/5)

We now analyze the access log of an Apache WebServer

```
public class LogAnalyzer {
    JavaRDD<String> logLines = sc.textFile(pathToFile);
    /* Convert the text log lines to ApacheAccessLog objects
    (cached, multiple transformations applied on those data) */
JavaRDD<ApacheAccessLog> accessLogs =
    logLines.map(line -> ApacheAccessLog.parseFromLogLine(line))
                                    .cache();

    // Calculate statistics based on the content size
contentSizeStats(accessLogs);
    // Compute Response Code to Count (take only the first 20)
    responseCodeCount(accessLogs);
    // Any IP that has accessed the server more than 100 times
    frequentClient(accessLogs, 100);
    // Top-K RequestedResources
    topKRequestedResources(accessLogs, 10);
}
```

This is only an excerpt

Example: LogAnalyzer (2/5)

```
private static void contentSizeStats(
                                        JavaRDD<ApacheAccessLog> accessLogs){

      JavaRDD<Long> contentSizes =
              accessLogs.map(log -> log.getContentSize()).cache();

      Long totalContentSize =
                               contentSizes.reduce((a, b) -> a + b);
      long numContentRequests = contentSizes.count();
      Long minContentSize =
                            contentSizes.min(Comparator.naturalOrder());
      Long maxContentSize =
                            contentSizes.max(Comparator.naturalOrder());

System.out.println("Content Size (byte):  average = "
            + totalContentSize / numContentRequests +
            ", minimum = " + +minContentSize +
            ", maximum = " + maxContentSize);
}
```

This is only an excerpt

Example: LogAnalyzer (3/5)

```
private static void responseCodeCount(
                                JavaRDD<ApacheAccessLog> accessLogs){

    JavaPairRDD<Integer, Long>  responseCodePairs =
        accessLogs.mapToPair(log ->
                new Tuple2<>(log.getResponseCode(), 1L));

    JavaPairRDD<Integer, Long>  responseCodeCounts =
        responseCodePairs.reduceByKey((a, b) -> a + b);

    List<Tuple2<Integer, Long>> responseCodeToCount =
        responseCodeCounts.take(20);

    System.out.println(
        String.format(
                "Response code counts: %s", responseCodeToCount
        )
    );
}
```

This is only an excerpt

Example: LogAnalyzer (4/5)

```
private static void frequentClient(
                    JavaRDD<ApacheAccessLog> accessLogs, int times){

    List<String> ipAddresses =
        accessLogs.mapToPair(
                    log -> new Tuple2<>(log.getIpAddress(), 1L))
        .reduceByKey((a, b) -> a + b)
        .filter(tuple -> tuple._2() > times)
        .map(tuple -> tuple._1())
        .collect();

    System.out.println(
        String.format(
          "IPAddresses > " + times + " times: %s", ipAddresses)
    );
}
```

This is only an excerpt

Example: LogAnalyzer (5/5)

```
private static void topKRequestedPDFs(
                                        JavaRDD<ApacheAccessLog> accessLogs,
                                        int k){

     List<Tuple2<String, Long>> topEndpoints = accessLogs
     .map(log -> log.getEndpoint())
     .filter(endpoint -> endpoint.toLowerCase().endsWith("pdf"))
     .mapToPair(endPoint -> new Tuple2<>(endPoint, 1L))
     .reduceByKey((a, b) -> a + b)

          // sort data and take the top k endpoints
.top(k, new ValueComparator( [...] )
          );
     [...]
}
```

This is only an excerpt

Spark SQL

Spark as unified engine

Spark SQL: example

- The dataset d14_filtered.csv contains recordings made at intervals of 20 seconds by sensors placed inside houses. Each line of the file has the format:

```
id, timestamp, value, property, plug_id,
                household_id, house_id
```

- Query1: locate houses with instant power consumption greater than or equal to 350 watts.

Q1: house filtering by instant power

```
public class  Query1Preprocessing {

public static JavaRDD<Tuple3<..>> preprocessDataset(JavaSparkContext sc) {

        JavaRDD<String> energyFile = sc.textFile(pathToFile);
        JavaRDD<Outlet> outlets =
        energyFile.map(line -> OutletParser.parseCSV(line))
                .filter(x -> x != null && x.getProperty().equals("1"));

        JavaRDD<Tuple3<String, String, Double>> result =
                        outlets.map(x -> new Tuple3<String, String, Double>
                (x.getHouse_id(), x.getTimestamp(), Double.parseDouble(x.getValue()))
                                                );

        return result;
    }
}
```

Q1: house filtering by instant power

```
SELECT
        house_id,
        SUM(value) AS sum

FROM query1

GROUP BY
        house_id,
        timestamp
```

→ temp →

```
SELECT    DISTINCT
        house_id

FROM temp

WHERE sum >= 350
```