Spark

Lightning-Fast Cluster Computing

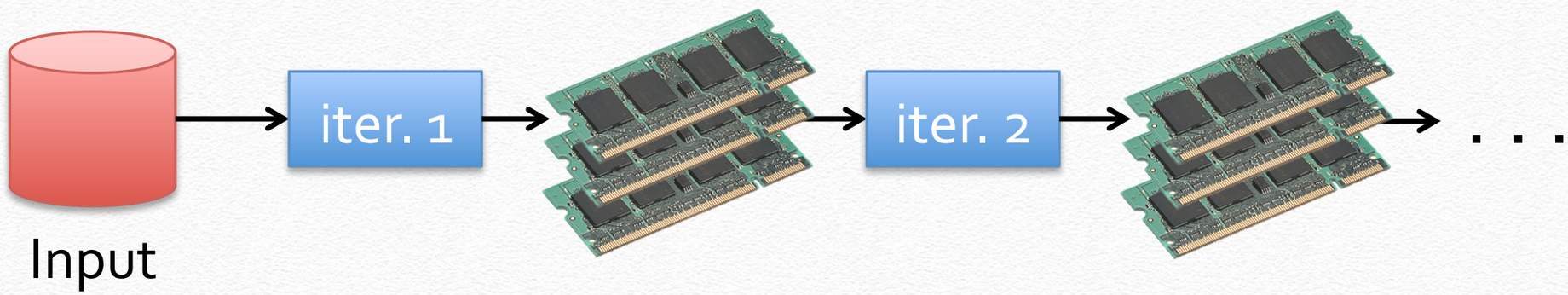
Core and SQL

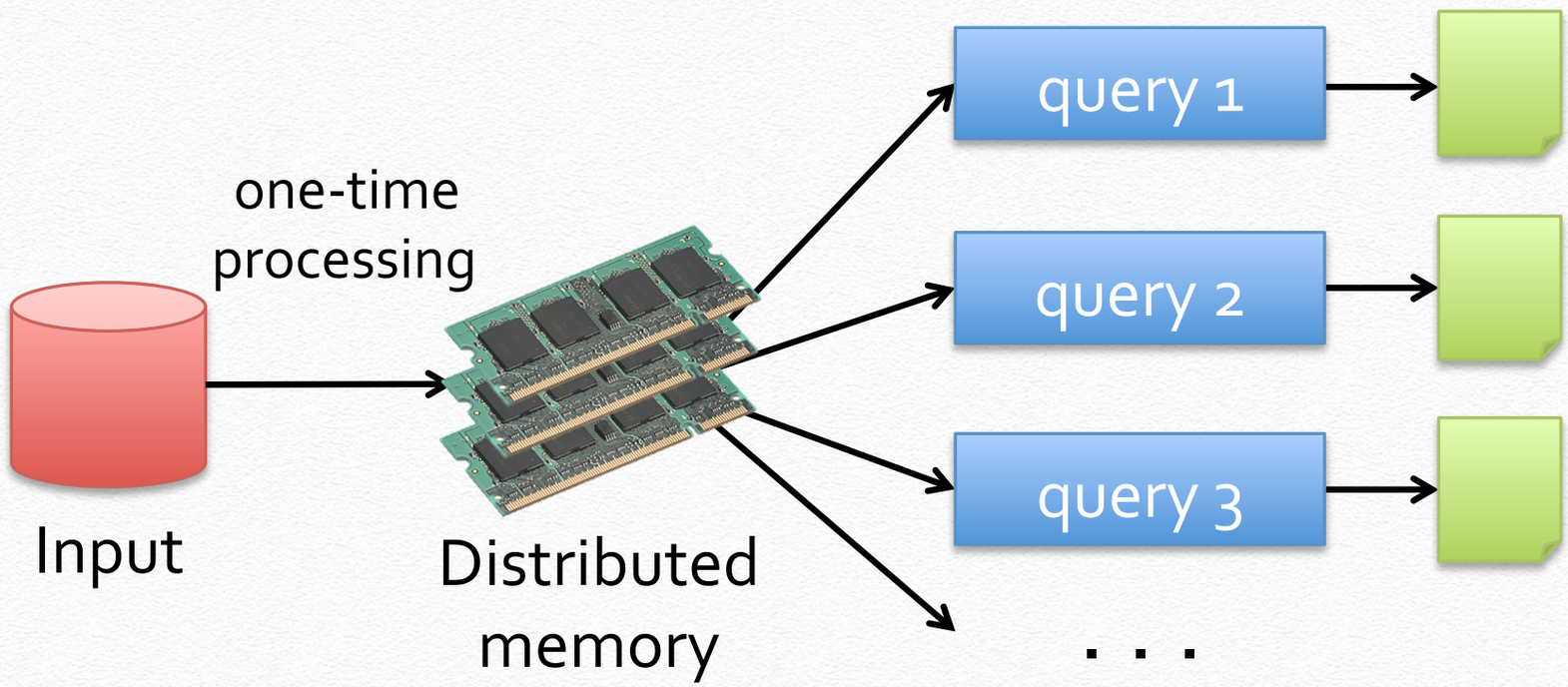
22/04/2020 - Big Data

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

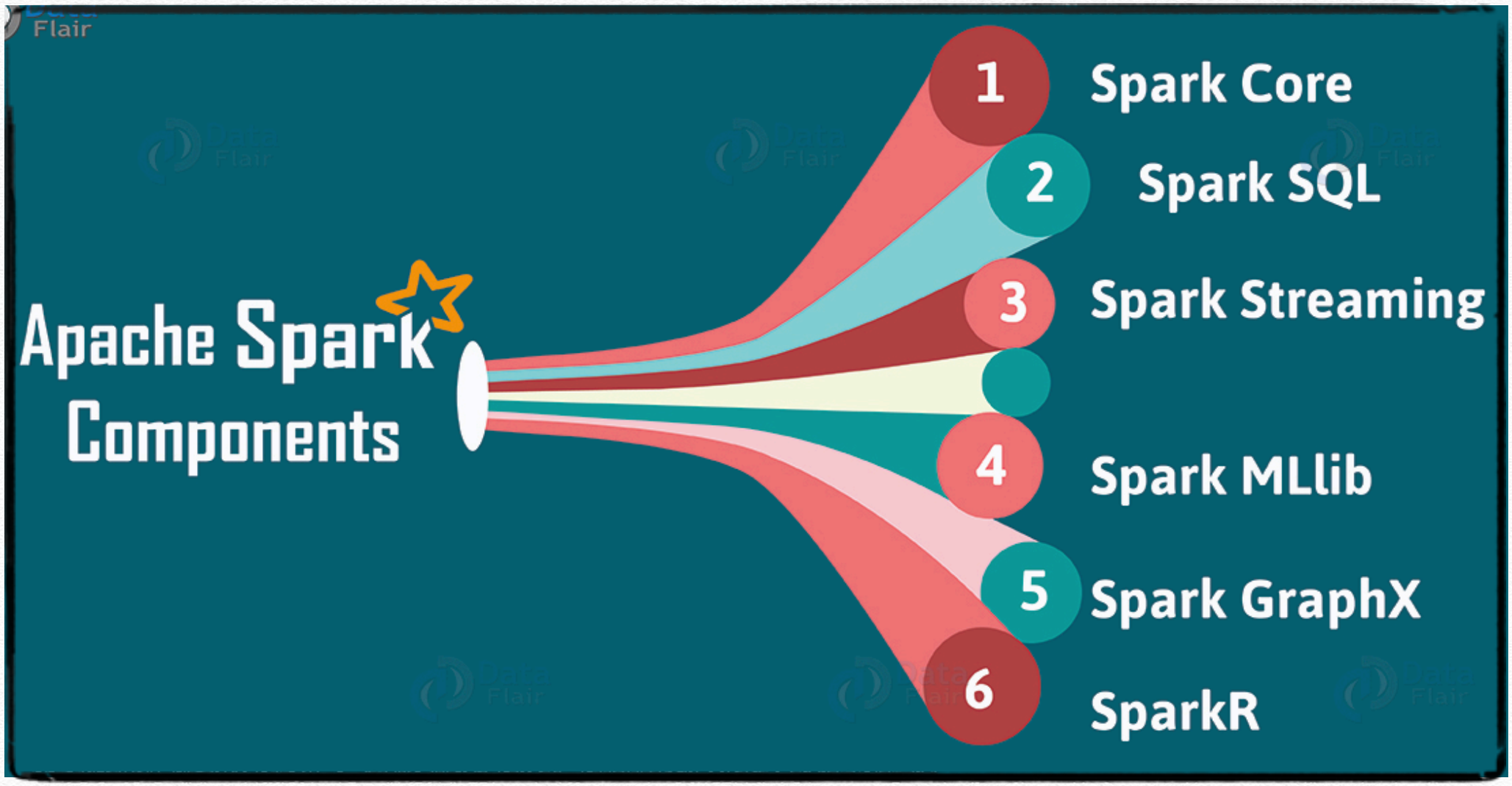
Apache Spark



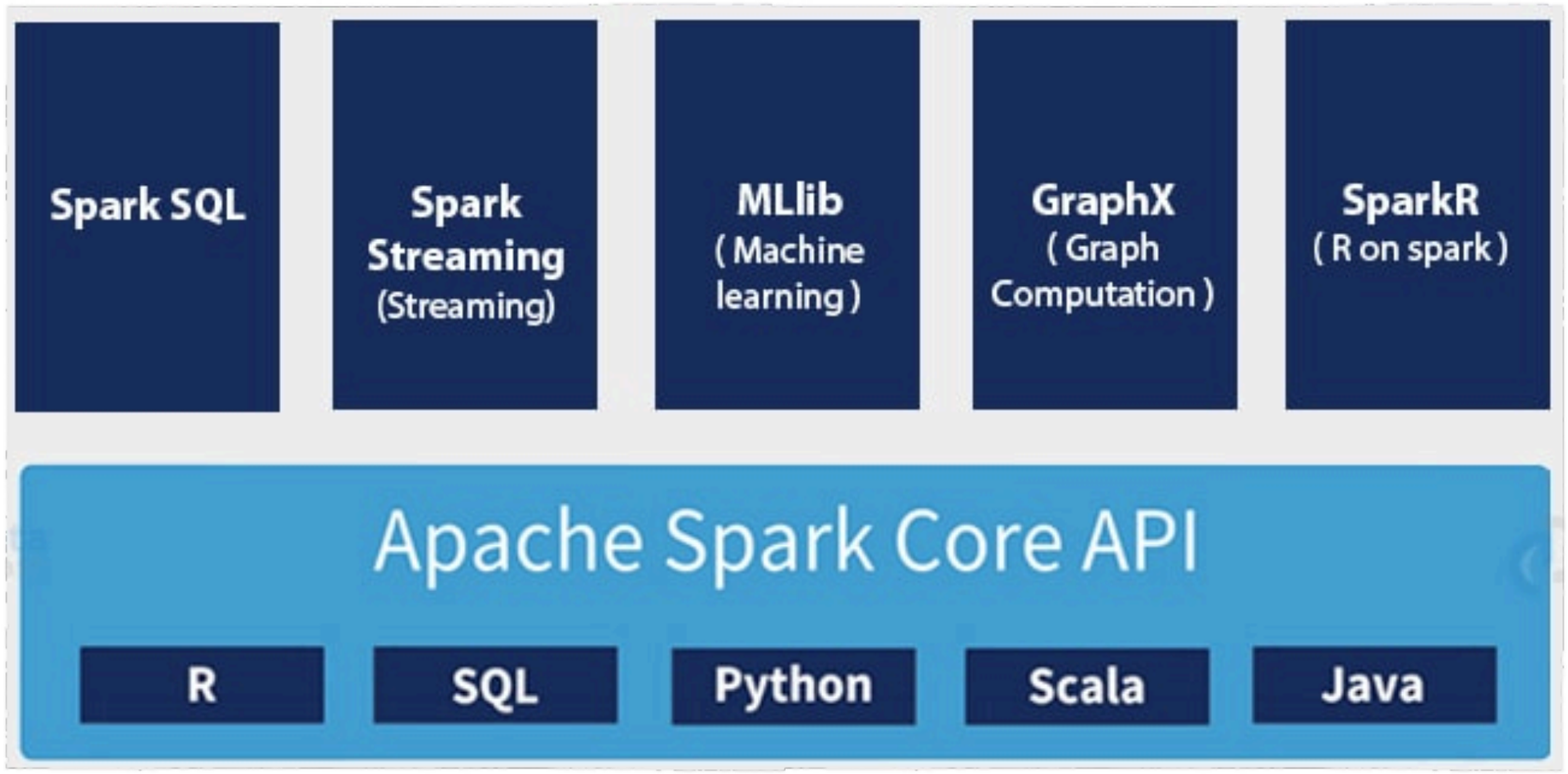


Spark

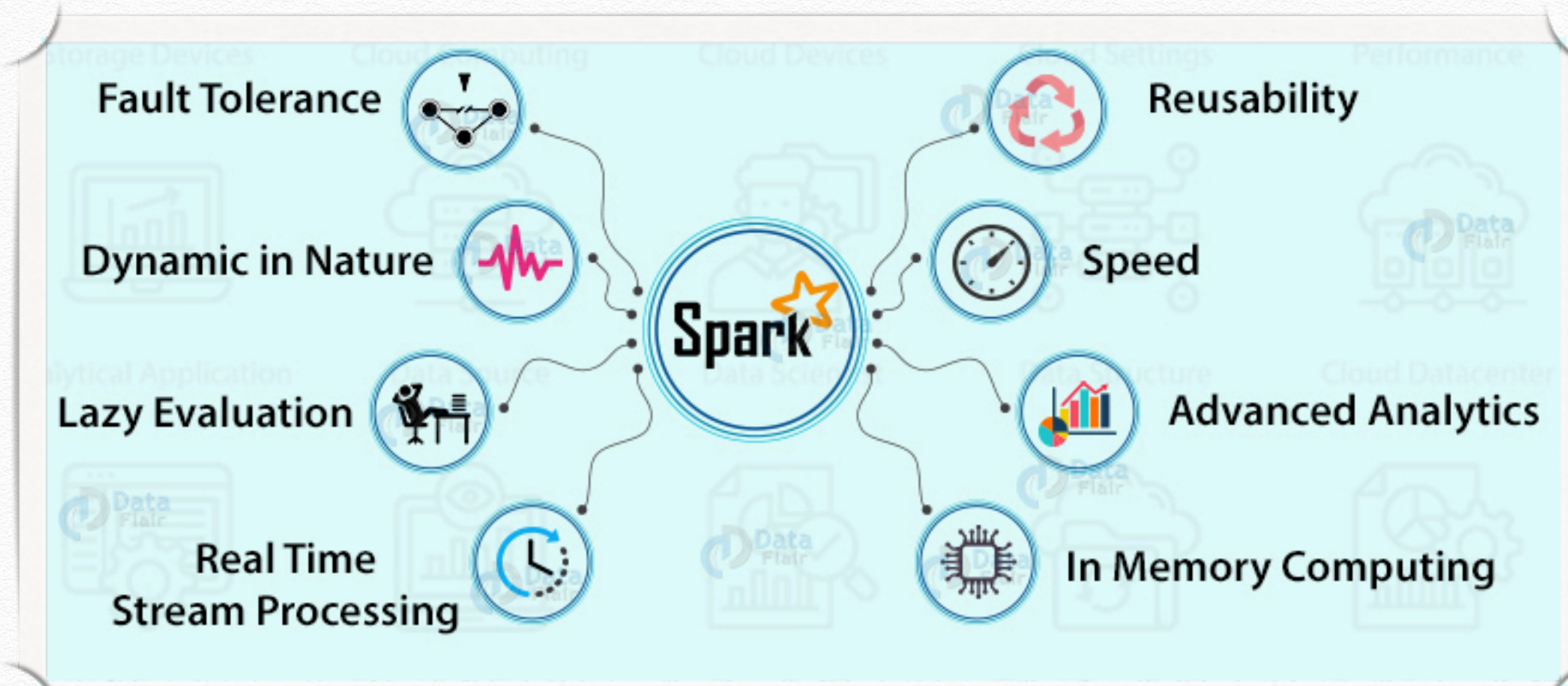
Combine SQL, streaming, and complex analytics



Apache Spark EcoSystem



Features


Fault Tolerance
Dynamic in Nature
Lazy Evaluation
Real Time
Stream Processing
Spark
Reusability
Speed
Advanced Analytics
In Memory Computing

Use Cases

- Finance Industry
- Healthcare
- Industry
- Media & Entertainment
- E-commerce Industry
- Travel Industry
- Data Streaming
- Machine Learning
- Fog Computing
- Interactive Analysis

Limitations

Spark Vs Hadoop

Factors	Spark	Hadoop MapReduce
Speed	100x times than MapReduce	Faster than traditional system
Written In	Scala	Java
Data Processing	Batch / real-time / iterative / interactive /graph	Batch processing
Ease of Use	Compact & easier than Hadoop	Complex & lengthy
Caching	Caches the data in-memory & enhances the system performance	Doesn't support caching of data

Spark

Apache Hive

VS

Spark SQL

- Apache **Hive** is built on top of ***Hadoop***.
- It is an open source data warehouse system.
- helps for analyzing and querying large datasets stored in Hadoop files.
- we have to write complex Map-Reduce jobs. But, using Hive, we just need to submit merely SQL queries.
- Users who are comfortable with SQL, Hive is mainly targeted towards them.

- In **Spark**, we use Spark SQL for structured data processing.
- We get more information of the structure of data by using SQL.
- Also, gives information on computations performed, that is, one can achieve extra optimization in Apache Spark, with this extra information.
- Although, Interaction with Spark SQL is possible in several ways, such as DataFrame and the Dataset API.

Spark Configuration

- Download a **binary release** of **apache Spark:**
- **spark-3.0.0-preview2-bin-hadoop3.2.tgz**

Download Apache Spark™

1. Choose a Spark release: 3.0.0-preview2 (Dec 23 2019)
2. Choose a package type: Pre-built for Apache Hadoop 3.2 and later
3. Download Spark: spark-3.0.0-preview2-bin-hadoop3.2.tgz
4. Verify this release using the 3.0.0-preview2 signatures, checksums and project release KEYS.

Note that, Spark is pre-built with Scala 2.11 except version 2.4.2, which is pre-built with Scala 2.12.

Environment variables

- In the **bash_profile** export all needed **environment variables**

```
Air-di-Roberto:~ roberto$ cd
Air-di-Roberto:~ roberto$ nano .bash_profile
```

```
  GNU nano 2.0.6              File: .bash_profile                        Modified

export PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
export JAVA_HOME=$(/usr/libexec/java_home)
export HADOOP_HOME=/Users/roberto/Documents/hadoop-3.2.1
export HIVE_HOME=/Users/roberto/Documents/apache-hive-2.3.6-bin
export SPARK_HOME=/Users/roberto/Documents/spark-3.0.0-preview2-bin-hadoop3.2
export PATH=$PATH:$HADOOP_HOME/bin:$HIVE_HOME/bin:$SPARK_HOME/bin
```

Mac OS X

Spark Running

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

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

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

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

- **Spark Shell - Scala**

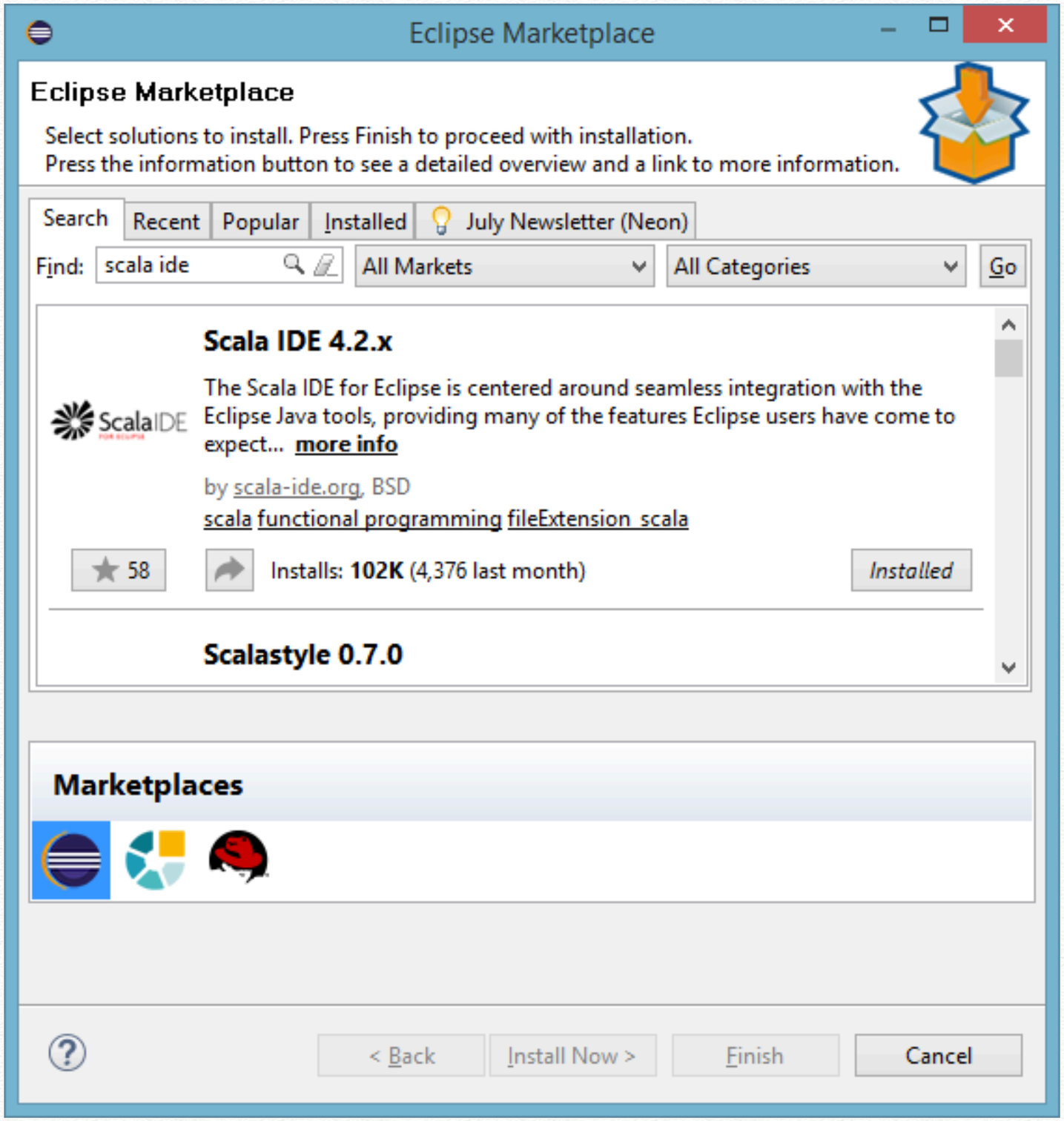
```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.0.0-preview2
      /_/

Using Scala version 2.12.10 (Java HotSpot(TM) 64-Bit Server VM, Java 13.0.1)

Type in expressions to have them evaluated.

Type :help for more information

scala>
```

Spark UI

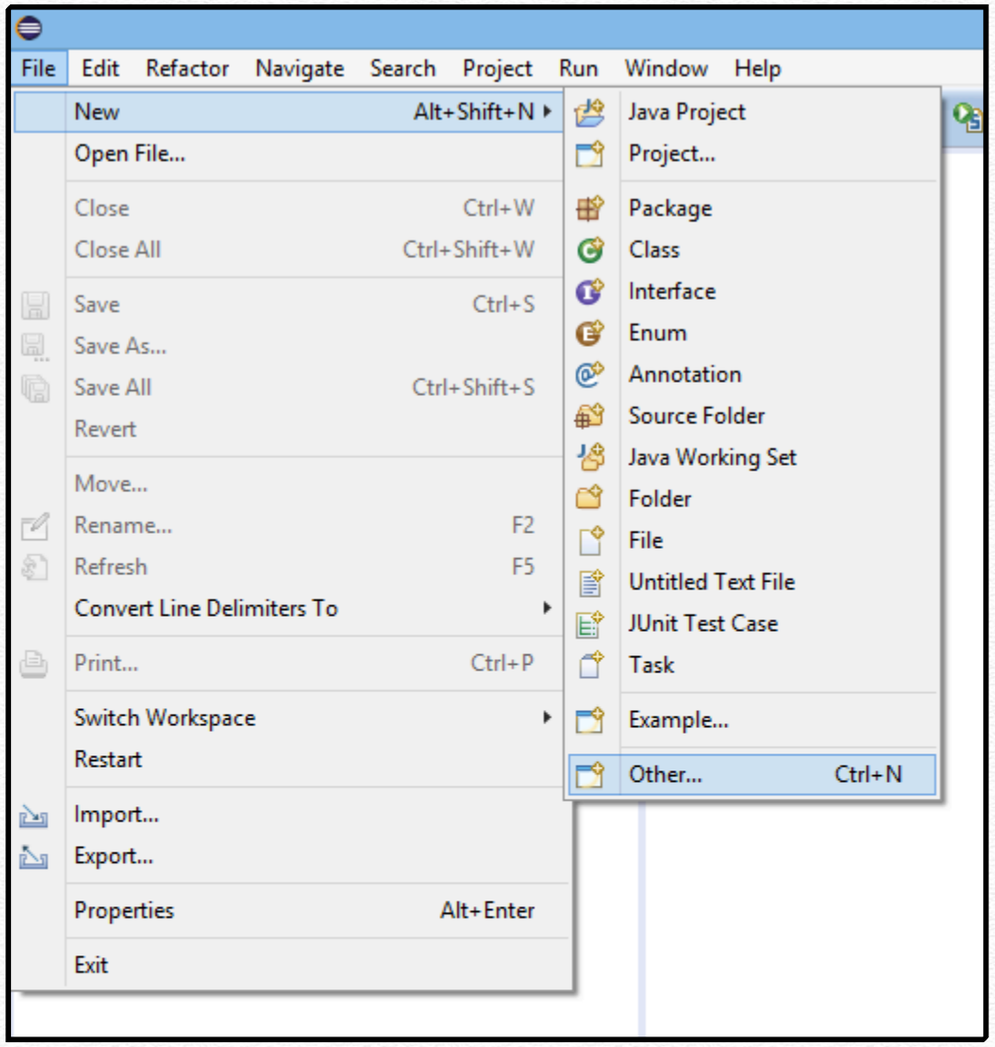
http://localhost:4040

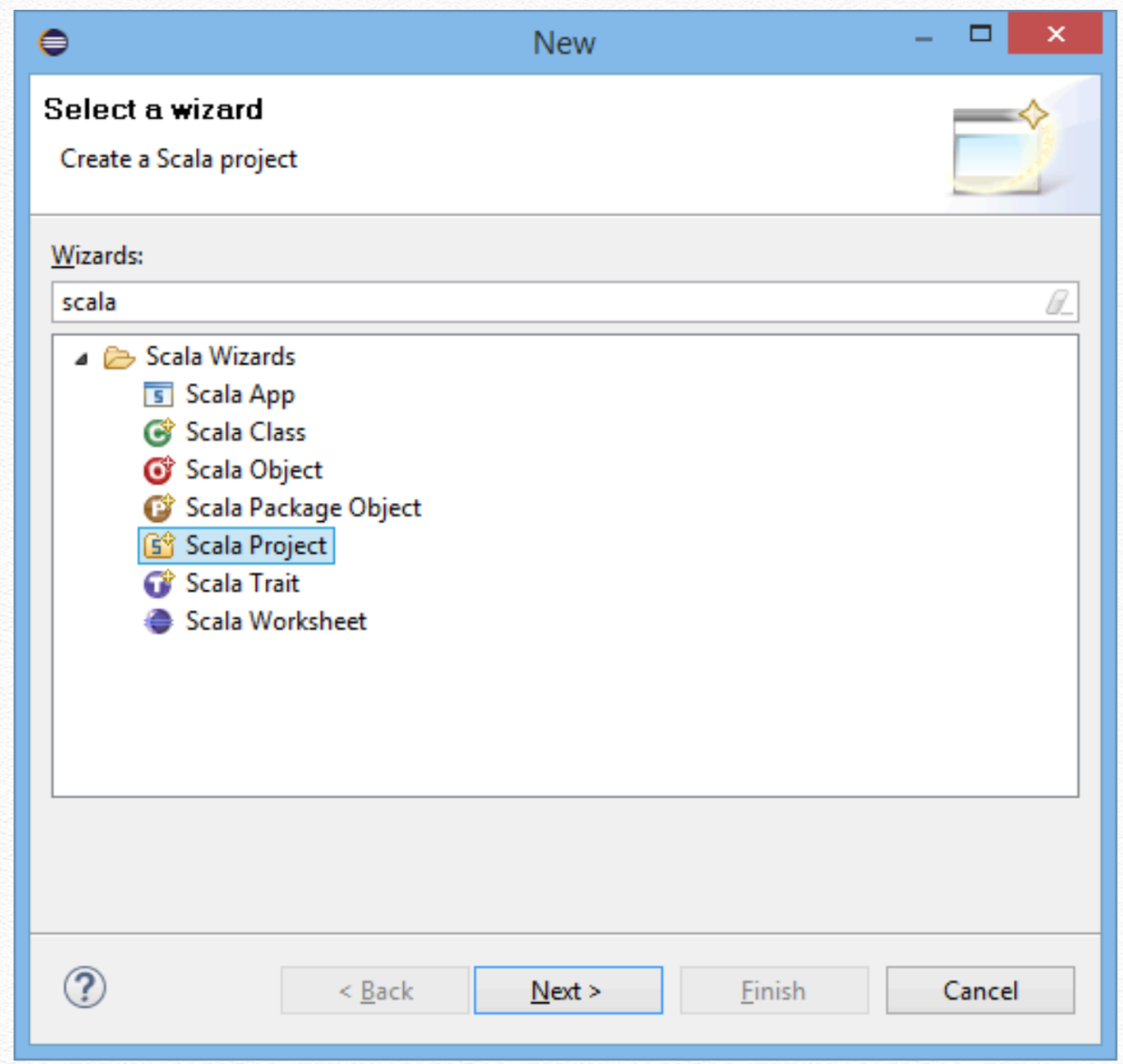
- This is the GUI for Spark Application, in local mode spark shell runs as an application. The GUI provide details about stages, storage (cached RDDs), Environment Variables and executors

Eclipse: SCALA



Eclipse: SCALA

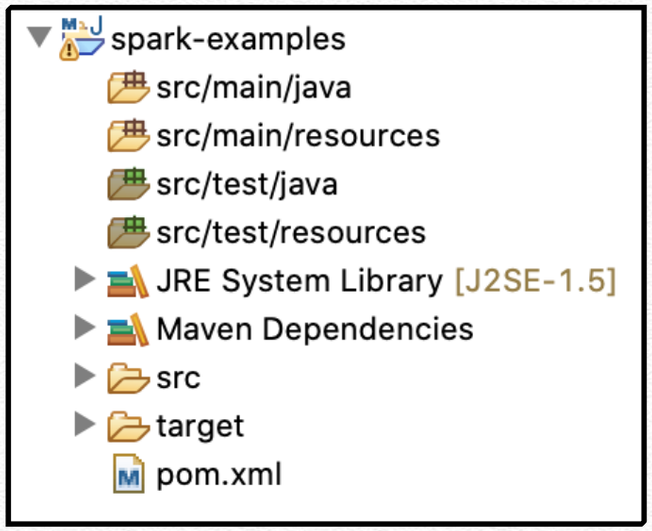




Python

Eclipse: JAVA

- **Maven Project**



- **pom.xml**

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://ww
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-examples</artifactId>
  <version>3.1.0</version>

  <packaging>jar</packaging>

  <name>Spark Project Examples</name>

  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-core_2.12</artifactId>
        <version>3.0.0-preview2</version>
    </dependency>

    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-sql_2.12</artifactId>
        <version>3.0.0-preview2</version>
    </dependency>
  </dependencies>
</project>
```

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

- **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 = "data/simpleLog.txt"; // you can use "YOUR_SPARK_HOME/README.md"

    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);

  }

}
```

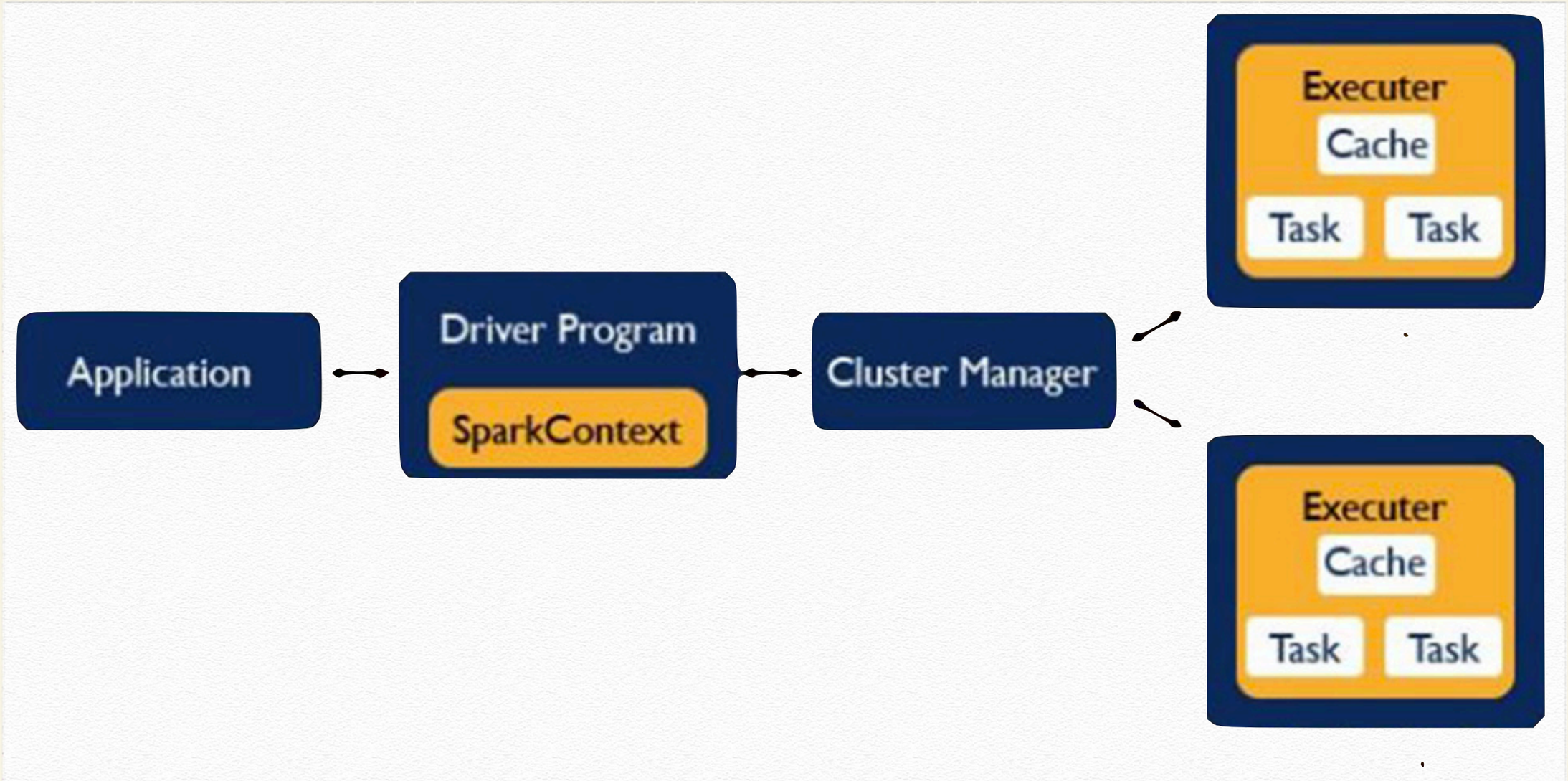
Spark Self-contained applications

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

```
String logFile = "data/simpleLog.txt";

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

JavaSparkContext sc = new JavaSparkContext(conf);

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

SparkContext is the entry point of Spark functionality. It allows your Spark Application to access Spark Cluster with the help of Resource Manager (standalone, Yarn, etc..)

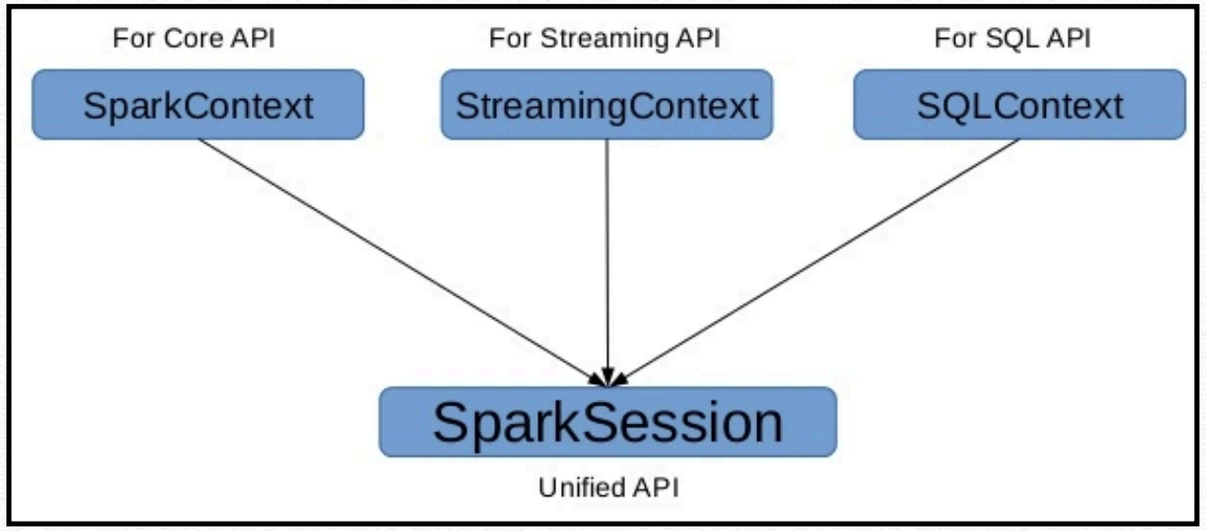
Spark Self-contained applications

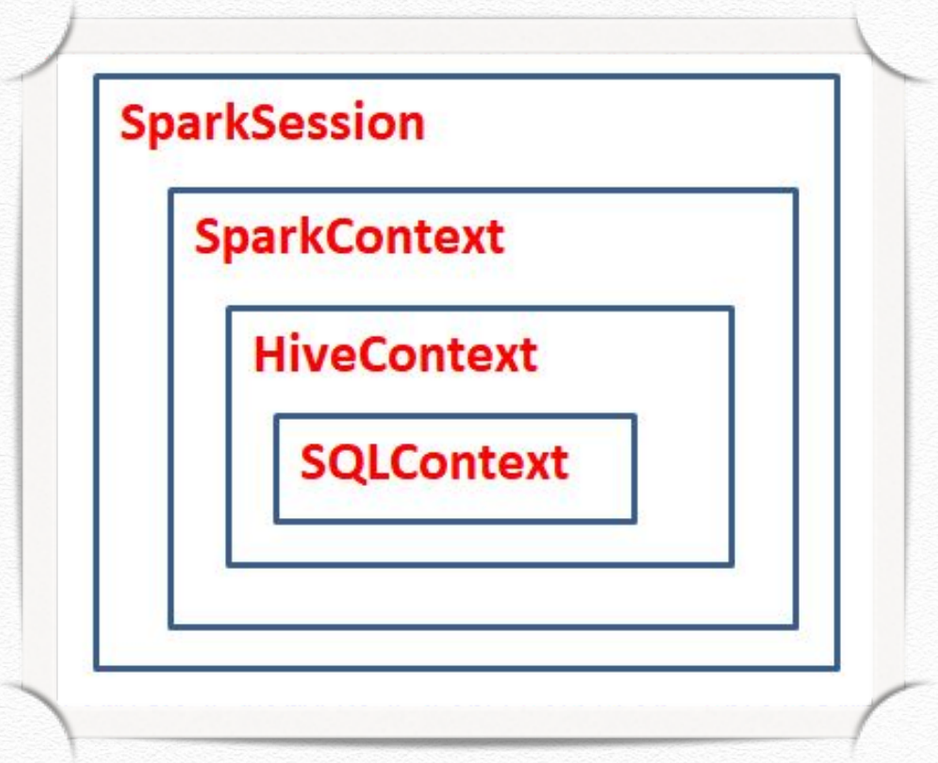


Spark Self-contained applications

- **Spark session** is a unified entry point of a spark application from Spark 2.0. It provides a way to interact with various spark's functionality with a lesser number of constructs.

- Instead of having a Spark Context, Hive Context, SQL context, now all of it is encapsulated in a Spark session.





Spark Self-contained applications

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

```
String logFile = "data/simpleLog.txt";

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

JavaSparkContext sc = new JavaSparkContext(conf);

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

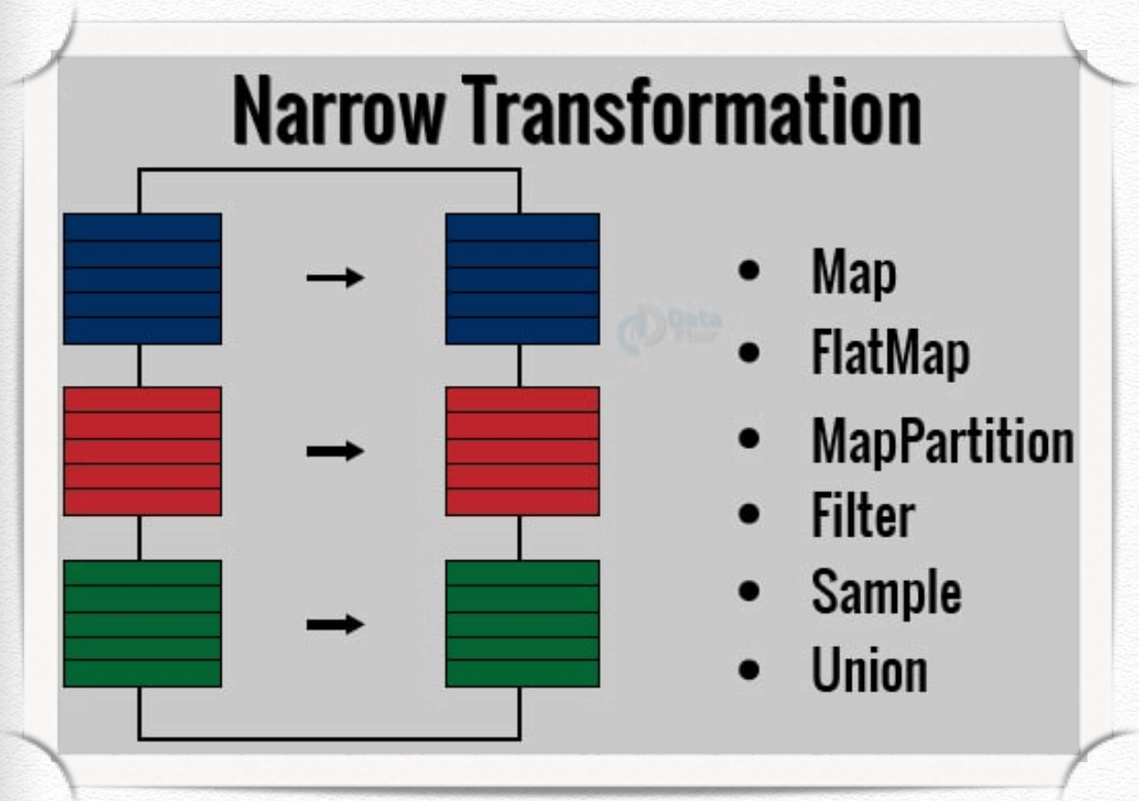
Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects.

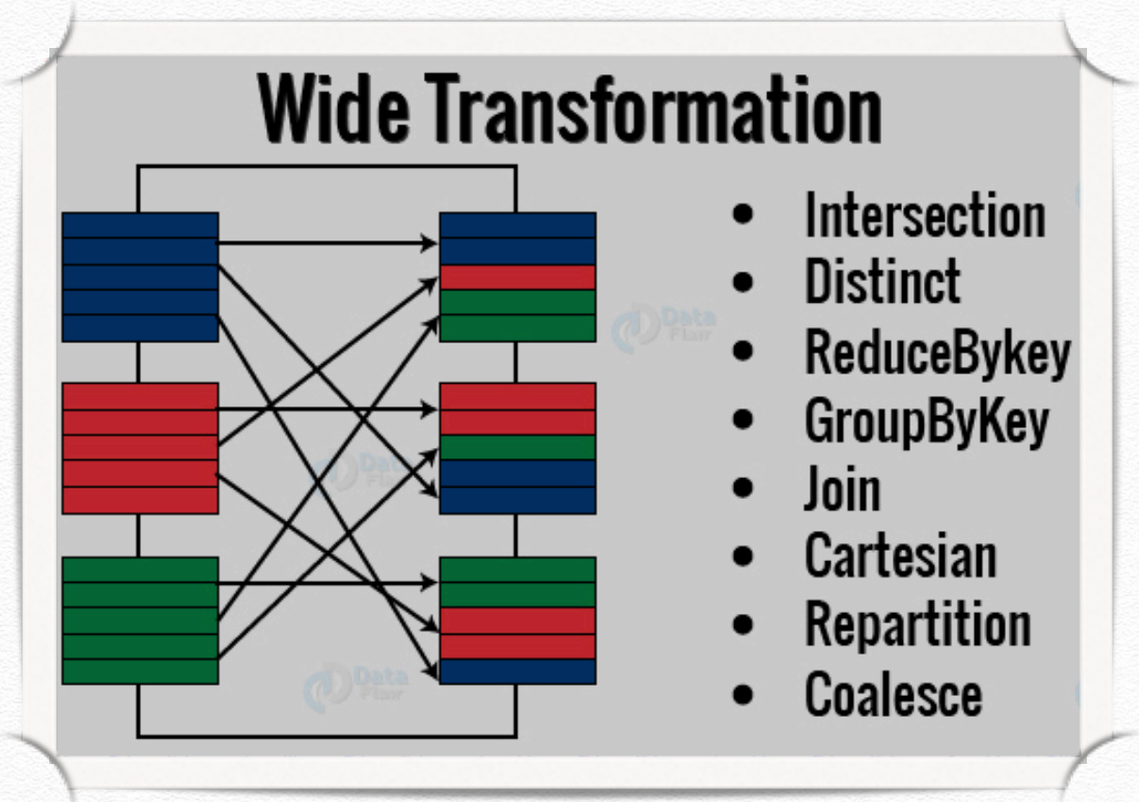
Spark RDD Operations

- RDD in Apache Spark supports two types of **operations**:
  - ***Transformations:*** *functions* that take an RDD as the input and produce one or many RDDs as the output.
  - ***Actions:*** they return final result of RDD computations

Spark RDD Transformations

- There are two kinds of transformations: *narrow transformation*, *wide transformation*.
  - ***narrow transformation:*** an output RDD has partitions with records that originate from a single partition in the parent RDD, i.e. it is self-sufficient. Spark groups narrow transformations as a stage known as **pipelining**
  - ***wide transformation:*** the data required to compute the records in a single partition may live in many partitions of the parent RDD. Wide transformations are also known as ***shuffle transformations*** because they may or may not depend on a shuffle

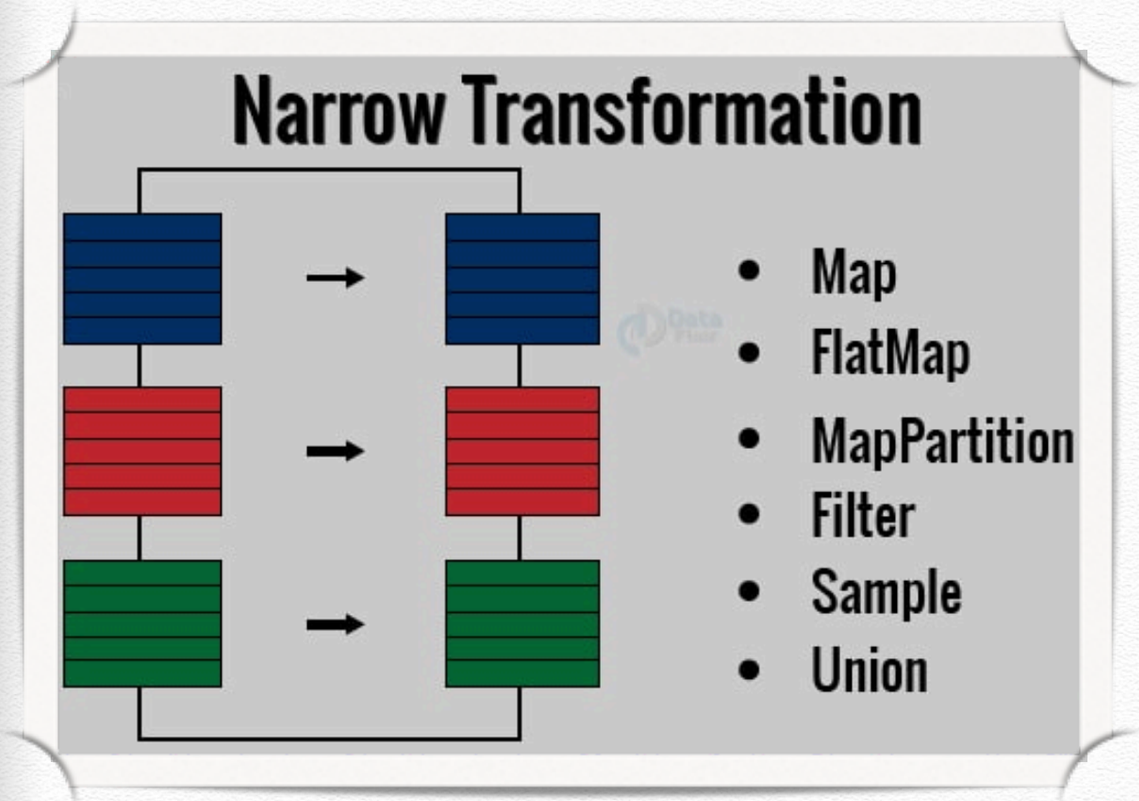


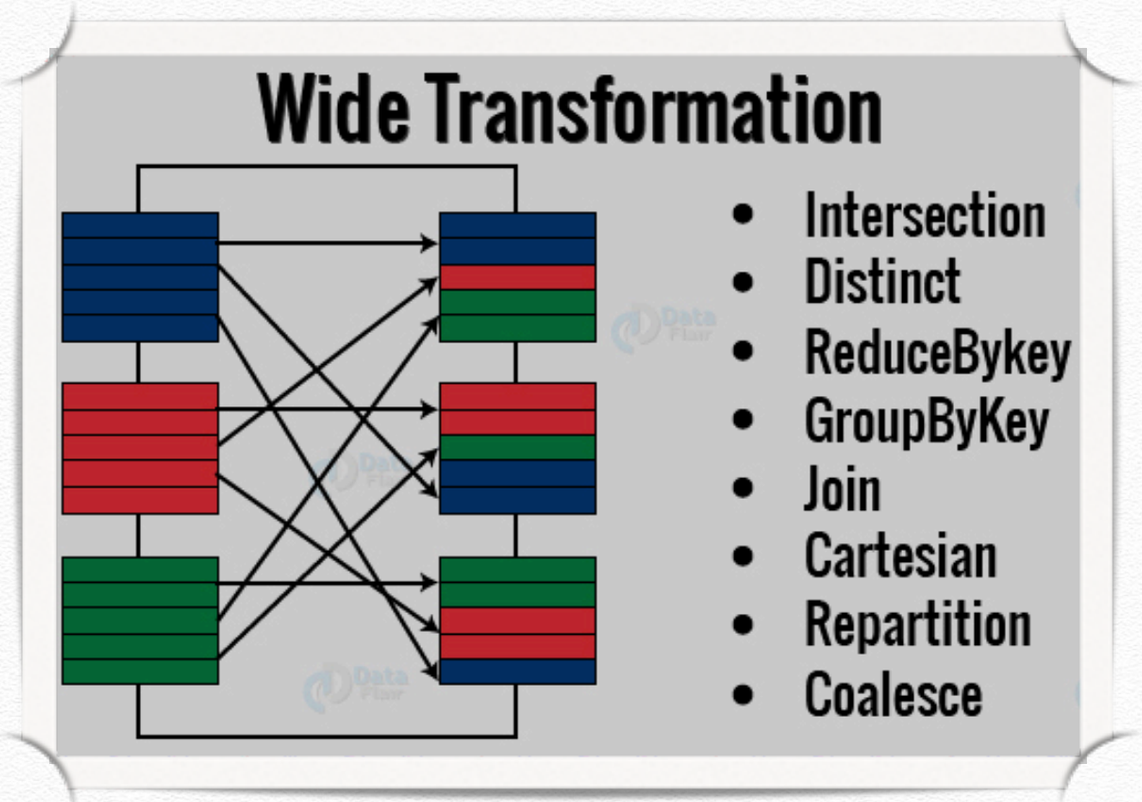


Spark RDD Actions

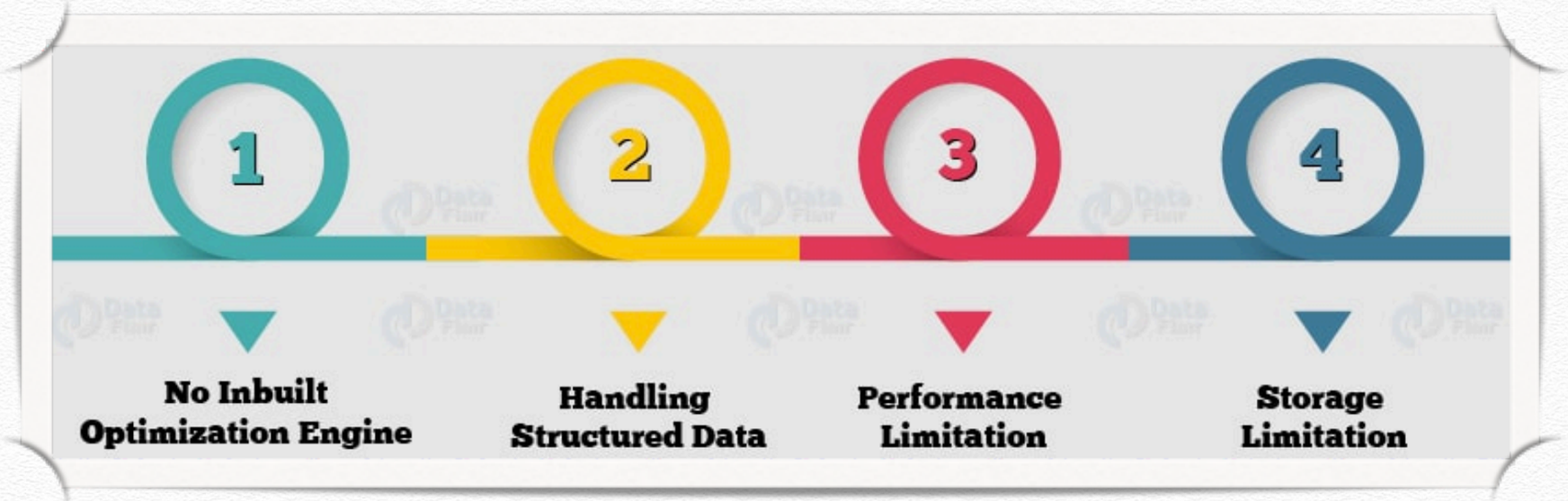
- An **Action** in Spark returns final result of RDD computations. It triggers execution using lineage graph to load the data into original RDD, carry out all intermediate transformations and return final results to Driver program or write it out to file system.

- **Actions** are RDD operations that produce **non-RDD values**. They materialize a value in a Spark program. An Action is one of the ways to send result from executors to the driver. ***First(), take(), reduce(), collect(), the count()*** is some of the Actions in spark.





Limitations of Spark RDD



- ***No inbuilt optimization engine:*** developers need to optimize each RDD based on its attributes.

- ***Handling structured data:*** RDDs don't infer the schema of the ingested data and requires the user to specify it.

- ***Performance limitation:*** being in-memory JVM objects, RDDs involve the overhead of Garbage Collection and Java Serialization which are expensive when data grows.

- ***Storage limitation:*** RDDs degrade when there is not enough memory to store them.

Spark Self-contained applications

- **Java Spark API: Spark actions**

Transformation

```
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);
```

Action

Spark Running - standalone

- Running Java Spark applications:

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

- **Parameters**

  - **–class**: To specify the class name to execute
  - **–master**: Master [***local / <Spark-URI> / yarn***]
  - **<Jar-Path>**: The jar file of application
  - **<Input-Path>**: Location from where input data will be read
  - **<Output-Path>**: Location where Spark application will write output

Spark Running - standalone

- Running Java Spark applications:

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

                --class "SimpleApp"

                --master local[4]

                SparkProject-1.0.jar
```

local to run locally with one thread, or local[N] to run locally with N threads.

- **output [terminal] - using simpleLog.txt**

```
Lines with a: 1, Lines with b: 0
```

Spark Running - standalone

- Running Java Spark applications:

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

                --class "SimpleApp"

                --master local[4]

                SparkProject-1.0.jar
```

local to run locally with one thread, or local[N] to run locally with N threads.

- **output [terminal] - using README.md**

```
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
```

The --master option allows to specify the master URL for a distributed cluster

- **output [terminal] - using README.md**

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

Spark Running - AWS

- Look at

http://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-launch.html

Lambda Expressions

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

Lambda Expressions

- Suppose that you are creating 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() {
        // ...
    }
}
```

Lambda Expressions

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

- Approach 1: Create Methods That Search for Members That Match One Characteristic

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

Lambda Expressions

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

- Approach 2: Create More Generalized Search Methods

```
public static void printPersonsWithinAgeRange(List<Person> roster,
                                              int low, int high) {
    for (Person p : roster) {
        if (low <= p.getAge() && p.getAge() < high) {
            p.printPerson();
        }
    }
}
```

Lambda Expressions

- Approach 3: Specify Search Criteria Code in a Local Class

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

interface CheckPerson {
    boolean test(Person p);
}

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

Lambda Expressions

- Approach 4: Specify Search Criteria Code in an Anonymous Class

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

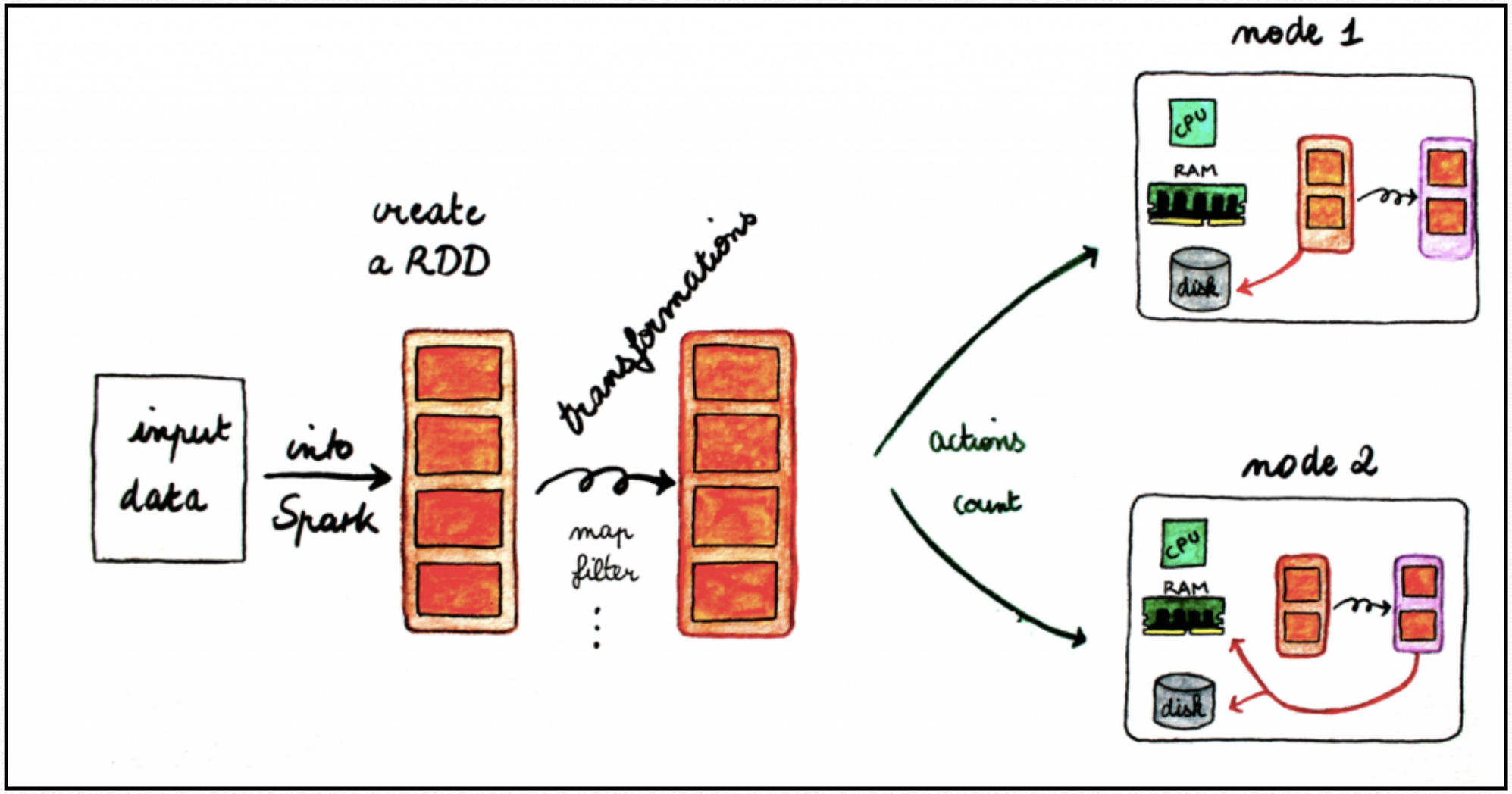
Lambda Expressions

- Approach 5: Specify Search Criteria Code with a Lambda Expression

```
printPersons(
    roster,
    (Person p) -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

Exercises

SPARK Core API



Exercises

SPARK Core API: Word Count

```
public class JavaWordCount {

  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);
    }

  // Spark Session creation

    SparkSession spark = SparkSession
      .builder()
      .appName("JavaWordCount")
      .getOrCreate();
```

Exercises

SPARK Core API: Word Count

```
// Import and Map creation

 JavaRDD<String> lines = spark.read().textFile(args[0]).javaRDD();

 JavaRDD<String> words = lines
                         .flatMap(s -> Arrays.asList(SPACE.split(s)).iterator());

// Reduce creation

  JavaPairRDD<String, Integer> ones = words.mapToPair(s -> new Tuple2<>(s, 1));

  JavaPairRDD<String, Integer> counts = ones.reduceByKey((i1, i2) -> i1 + i2);
```

Exercises

SPARK Core API: Word Count

```
/**
   *  Now just keep the word which appear strictly more than x times!
   */

 public JavaPairRDD<String, Integer> filterOnWordcount(
                                JavaPairRDD<String, Integer> wordcounts, int x) {

   JavaPairRDD<String, Integer> filtered =
                              wordcounts.filter(couple -> couple._2() > x);

   return filtered;

}
```

Exercises

SPARK Core API: Word Count

```
// Output Result

 List<Tuple2<String, Integer>> output = counts.collect();

 for (Tuple2<?,?> tuple : output) {
    System.out.println(tuple._1() + ": " + tuple._2());
  }

 // Spark Session termination

    spark.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**

```
import com.fasterxml.jackson.databind.ObjectMapper;

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)**

```
  - Find all the persons mentioned on tweets

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)

```
    - Find all the persons mentioned on tweets


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

    JavaRDD<String> mentions =
         tweets.flatMap(tweet -> Arrays.asList(tweet.getText()
               .split(" “)).iterator())
               .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)**

```
  - Count how many times each person is mentioned

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)**

```
  - Count how many times each person is mentioned


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)**

```
  - Find the 10 most mentioned persons by descending order


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)**

```
  - Find the 10 most mentioned persons by descending order


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

SPARK Core API: Read and Write with HDFS

```
public class JavaReadWriteHDFS {

    public static void main(String[] args) {

        if (args.length < 2) {
          System.err.println("Please provide the input file and output full path as argument");
          System.exit(0);
        }

        SparkConf conf = new SparkConf().setAppName("java.ReadFromHdfs").setMaster("local");
        JavaSparkContext context = new JavaSparkContext(conf);

        JavaRDD<String> textFile = context.textFile(args[0]);
```

The "args[0]" can be a single text file or a directory that contains multiple files. The "JavaRDD" will contain individual records from the file(s). Useful to process CSV files that can be split by records.

args[0] can be

1) A file: hdfs://localhost:9000/sampledata/sample.txt

2) a folder with many files: hdfs://localhost:8020/sampledata

Exercises

SPARK Core API: Read and Write with HDFS

```
        List<String> listCsvLines = textFile.collect();
         System.out.println("========= " + listCsvLines);



        textFile.saveAsTextFile(args[1]);

        context.close();
    }
}
```

args[1] can be a folder: hdfs://localhost:9000/out

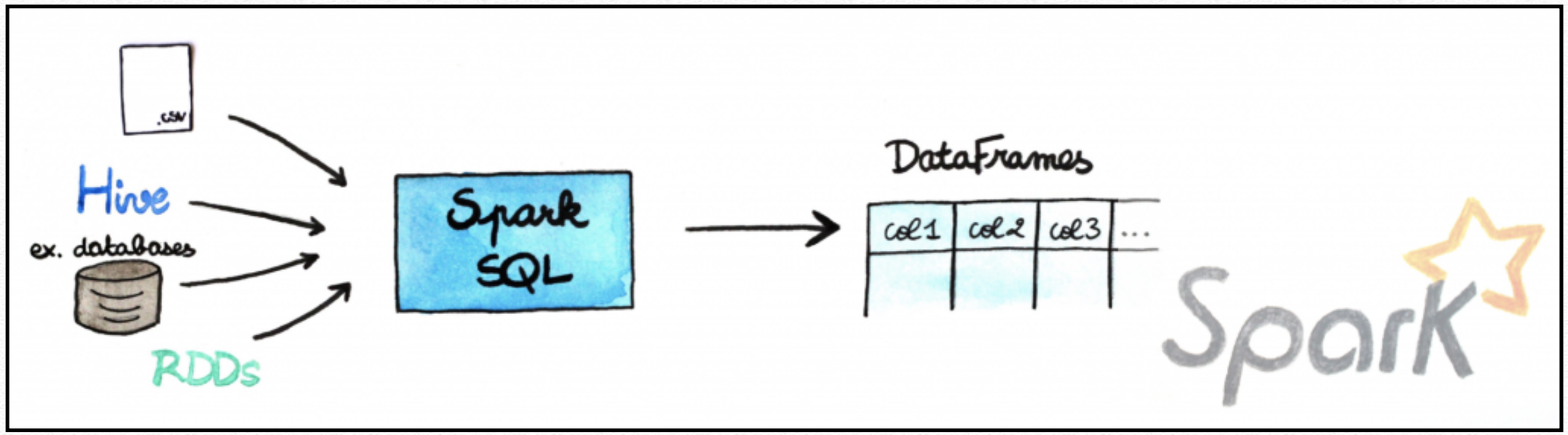
Exercises

SPARK Core API: Read and Write with HDFS

```
        List<String> listCsvLines = textFile.collect();
        System.out.println("========= " + listCsvLines);

        /*
        args[1] can be a folder: hdfs://localhost:9000/out

         */

        textFile.saveAsTextFile(args[1]);

        context.close();
    }
}
```

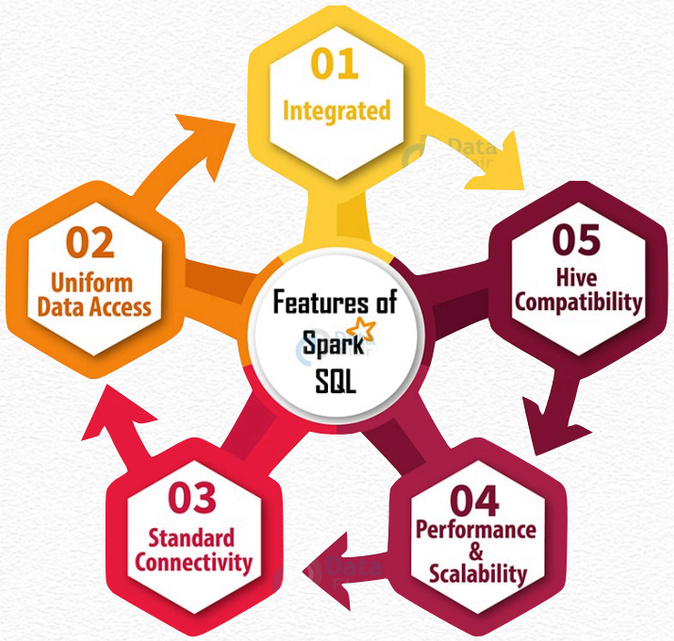
Exercises

- **SPARK SQL (DataFrame)**



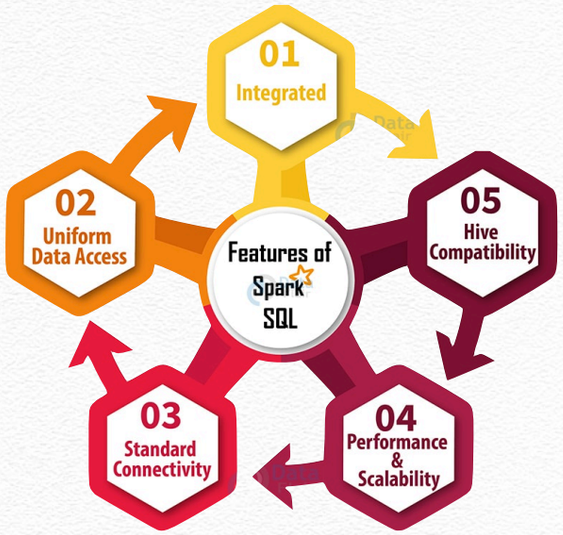
What is Apache Spark SQL

- **Apache Spark SQL** integrates relational processing with Sparks functional programming.
- It is Spark module for structured data processing. Spark SQL blurs the line between **RDD** and relational table.
- Spark SQL runs on top of the Spark Core
- **DataFrame** API and **Datasets** API are the ways to *interact* with Spark SQL



Sppark SQL DataFrame and Dataset

- **DataFrame** is a *Dataset* organized into named columns
- DataFrames are similar to the table in a relational database
- DataFrame contains rows with **Schema**. The schema is the illustration of the structure of data
- DataFrame in Apache Spark prevails over RDD but contains the features of RDD as well
- **Dataset** is a data structure in SparkSQL which is strongly typed and is a map to a relational schema
- we represent DataFrame as Dataset of rows: in Java API, the user uses **Dataset<Row>** to represent a DataFrame.



Exercises

SPARK DataFrame: SparkSQL

```
public class JavaSparkSQLExample {
  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 {

        SparkSession spark = SparkSession
                            .builder()
                            .appName("Java Spark SQL basic example")
                            .config("spark.some.config.option", "some-value")
                            .getOrCreate();

        runBasicDataFrameExample(spark);
        runDatasetCreationExample(spark);
        runInferSchemaExample(spark);
        runProgrammaticSchemaExample(spark);

        spark.stop();
}
```

To all the functionality of Spark, SparkSession class is the entry point. For the creation of basic SparkSession just use `SparkSession.builder()`
Using Spark Session, an application can create DataFrame from an existing RDD, Hive table or from Spark data sources

Exercises

people.json

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

SPARK DataFrame: SparkSQL

```
private static void runBasicDataFrameExample(SparkSession spark)
                                                    throws AnalysisException {

 Dataset<Row> df = spark.read().json("data/people.json");

 // Displays the content of the DataFrame to stdout
 df.show();

 // +----+-------+
 // | age|   name|
 // +----+-------+
 // |null|Michael|
 // |  30|   Andy|
 // |  19| Justin|
 // +----+-------+

 // Print the schema in a tree format
 df.printSchema();

 // root
 // |-- age: long (nullable = true)
 // |-- name: string (nullable = true)
```

Exercises

people.json

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

SPARK DataFrame: SparkSQL

```
// Select only the "name" column
df.select("name").show();

// +-------+
// |   name|
// +-------+
// |Michael|
// |   Andy|
// | Justin|
// +-------+

// Select everybody, but increment the age by 1
df.select(col("name"), col("age").plus(1)).show();

// +-------+---------+
// |   name|(age + 1)|
// +-------+---------+
// |Michael|     null|
// |   Andy|       31|
// | Justin|       20|
// +-------+---------+
```

Exercises

people.json

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

SPARK DataFrame: SparkSQL

```
// Select people older than 21
 df.filter(col("age").gt(21)).show();

 // +---+----+
 // |age|name|
 // +---+----+
 // | 30|Andy|
 // +---+----+

 // Count people by age
 df.groupBy("age").count().show();

 // +----+-----+
 // | age|count|
 // +----+-----+
 // |  19|    1|
 // |null|    1|
 // |  30|    1|
 // +----+-----+
```

Exercises

people.json

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

SPARK DataFrame: SparkSQL

```
// Register the DataFrame as a SQL temporary view
 df.createOrReplaceTempView("people");

 Dataset<Row> sqlDF = spark.sql("SELECT * FROM people");
 sqlDF.show();

 // +----+-------+
 // | age|   name|
 // +----+-------+
 // |null|Michael|
 // |  30|   Andy|
 // |  19| Justin|
 // +----+-------+

}
```

Exercises

SPARK DataFrame: SparkSQL

```
private static void runDatasetCreationExample(SparkSession spark) {

 // Create an instance of a Bean class
 Person person = new Person();
 person.setName("Andy");
 person.setAge(32);

 // Encoders are created for Java beans
 Encoder<Person> personEncoder = Encoders.bean(Person.class);

 Dataset<Person> javaBeanDS = spark.createDataset(
   Collections.singletonList(person),
   personEncoder
 );

 javaBeanDS.show();
 // +---+----+
 // |age|name|
 // +---+----+
 // | 32|Andy|
 // +---+----+
```

Exercises

SPARK DataFrame: SparkSQL

```
// Encoders for most common types are provided in class Encoders
 Encoder<Integer> integerEncoder = Encoders.INT();

 Dataset<Integer> primitiveDS =
                     spark.createDataset(Arrays.asList(1, 2, 3), integerEncoder);

Dataset<Integer> transformedDS = primitiveDS
            .map(
             (MapFunction<Integer, Integer>) value -> value + 1, integerEncoder
            );

transformedDS.collect(); // Returns [2, 3, 4]
```

Exercises

people.json

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

SPARK DataFrame: SparkSQL

```
// DataFrames can be converted to a Dataset by providing a class.
// Mapping based on name

 String path = "data/people.json";

 Dataset<Person> peopleDS = spark.read().json(path).as(personEncoder);

 peopleDS.show();

 // +----+-------+
 // | age|   name|
 // +----+-------+
 // |null|Michael|
 // |  30|   Andy|
 // |  19| Justin|
 // +----+-------+


}
```

Exercises

people.txt

Michael, 29
Andy, 30
Justin, 19

SPARK DataFrame: SparkSQL

```
private static void runInferSchemaExample(SparkSession spark) {

 // Create an RDD of Person objects from a text file
 JavaRDD<Person> peopleRDD = spark.read()
   .textFile("data/people.txt")
   .javaRDD()
   .map(line -> {
     String[] parts = line.split(",");
     Person person = new Person();
     person.setName(parts[0]);
     person.setAge(Integer.parseInt(parts[1].trim()));
     return person;
   });

 // Apply a schema to an RDD of JavaBeans to get a DataFrame
 Dataset<Row> peopleDF = spark.createDataFrame(peopleRDD, Person.class);

 // Register the DataFrame as a temporary view
 peopleDF.createOrReplaceTempView("people");
```

Exercises

people.txt

Michael, 29
Andy, 30
Justin, 19

SPARK DataFrame: SparkSQL

```
// SQL statements can be run by using the sql methods provided by spark
 Dataset<Row> teenagersDF = spark
                        .sql("SELECT name FROM people WHERE age BETWEEN 13 AND 19");

 // The columns of a row in the result can be accessed by field index
 Encoder<String> stringEncoder = Encoders.STRING();

 Dataset<String> teenagerNamesByIndexDF = teenagersDF
       .map(
           (MapFunction<Row, String>) row -> "Name: " + row.getString(0),  stringEncoder
           );

 teenagerNamesByIndexDF.show();

 // +------------+
 // |       value|
 // +------------+
 // |Name: Justin|
 // +------------+
```

Exercises

people.txt

Michael, 29
Andy, 30
Justin, 19

SPARK DataFrame: SparkSQL

```
// or by field name

 Dataset<String> teenagerNamesByFieldDF = teenagersDF
      .map(
          (MapFunction<Row, String>) row -> "Name: " + row.<String>getAs("name"), stringEncoder
          );

 teenagerNamesByFieldDF.show();

 // +------------+
 // |       value|
 // +------------+
 // |Name: Justin|
 // +------------+

}
```

Exercises

people.txt

Michael, 29
Andy, 30
Justin, 19

SPARK DataFrame: SparkSQL

```
private static void runProgrammaticSchemaExample(SparkSession spark) {

 // Create an RDD
 JavaRDD<String> peopleRDD = spark.sparkContext()
   .textFile("data/people.txt", 1)
   .toJavaRDD();

 // The schema is encoded in a string
 String schemaString = "name age";

 // Generate the schema based on the string of schema
 List<StructField> fields = new ArrayList<>();

 for (String fieldName : schemaString.split(" ")) {
   StructField field = DataTypes
                       .createStructField(fieldName, DataTypes.StringType, true);
   fields.add(field);
 }

 StructType schema = DataTypes.createStructType(fields);
```

The `textFile` method also takes an optional second argument for controlling the number of partitions of the file

Exercises

people.txt

Michael, 29
Andy, 30
Justin, 19

SPARK DataFrame: SparkSQL

```
// Convert records of the RDD (people) to Rows
JavaRDD<Row> rowRDD = peopleRDD
                    .map((Function<String, Row>) record -> {
                                    String[] attributes = record.split(",");
                 return RowFactory.create(attributes[0], attributes[1].trim());
                     });

// Apply the schema to the RDD
Dataset<Row> peopleDataFrame = spark.createDataFrame(rowRDD, schema);

// Creates a temporary view using the DataFrame
peopleDataFrame.createOrReplaceTempView("people");
```

Exercises

people.txt

Michael, 29
Andy, 30
Justin, 19

SPARK DataFrame: SparkSQL

```
// SQL can be run over a temporary view created using DataFrames
Dataset<Row> results = spark.sql("SELECT name FROM people");

// 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 field index or by field name

 Dataset<String> namesDS = results.map(
     (MapFunction<Row, String>) row -> "Name: " + row.getString(0),
     Encoders.STRING());

 namesDS.show();

 // +-------------+
 // |        value|
 // +-------------+
 // |Name: Michael|
 // |Name: Andy   |
 // |Name: Justin |
 // +-------------+

}
```

Exercises

SPARK DataFrame: SparkSQL and HIVE

```
public class JavaSparkHiveExample {

public static class Record implements Serializable {
 private int key;
 private String value;

 public int getKey() {
   return key;
 }

 public void setKey(int key) {
   this.key = key;
 }

 public String getValue() {
   return value;
 }

 public void setValue(String value) {
   this.value = value;
 }
}
```

Exercises

kv1.txt

238	val_238
86	val_86
311	val_311

SPARK DataFrame: SparkSQL and HIVE

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

 // warehouseLocation points to the default location for managed databases and tables
 String warehouseLocation = new File("spark-warehouse").getAbsolutePath();

SparkSession spark = SparkSession
   .builder()
   .appName("Java Spark Hive Example")
   .config("spark.sql.warehouse.dir", warehouseLocation)
   .enableHiveSupport()
   .getOrCreate();

 spark.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING) USING hive");
 spark.sql("LOAD DATA LOCAL INPATH 'data/kv1.txt' INTO TABLE src");
```

Exercises

kv1.txt

238	val_238
86	val_86
311	val_311

SPARK DataFrame: SparkSQL and HIVE

```
// Queries are expressed in HiveQL

 spark.sql("SELECT * FROM src”).show();

 // +---+-------+
 // |key|  value|
 // +---+-------+
 // |238|val_238|
 // | 86| val_86|
 // |311|val_311|
 // ...

 // Aggregation queries are also supported.

 spark.sql("SELECT COUNT(*) FROM src").show();
 // +--------+
 // |count(1)|
 // +--------+
 // |    500 |
 // +--------+
```

Exercises

kv1.txt

238	val_238
86	val_86
311	val_311

SPARK DataFrame: SparkSQL and HIVE

```
// The results of SQL queries are themselves DataFrames and support all normal functions.
 Dataset<Row> sqlDF = spark
                 .sql("SELECT key, value FROM src WHERE key < 10 ORDER BY key");

 // The items in DataFrames are of type Row, which lets you to access each column by ordinal.
 Dataset<String> stringsDS = sqlDF
       .map(
        (MapFunction<Row, String>) row -> "Key: " + row.get(0) + ", Value: " + row.get(1),
         Encoders.STRING()
        );
 stringsDS.show();

 // +--------------------+
 // |               value|
 // +--------------------+
 // |Key: 0, Value: val_0|
 // |Key: 0, Value: val_0|
 // |Key: 0, Value: val_0|
 // |Key: 2, Value: val_2|
 // |Key: 4, Value: val_4|
 // ...
```

Exercises

kv1.txt

238	val_238
86	val_86
311	val_311

SPARK DataFrame: SparkSQL and HIVE

```
// You can also use DataFrames to create temporary views within a SparkSession.
 List<Record> records = new ArrayList<>();
 for (int key = 1; key < 100; key++) {
   Record record = new Record();
   record.setKey(key);
   record.setValue("val_" + key);
   records.add(record);
 }
 Dataset<Row> recordsDF = spark.createDataFrame(records, Record.class);
 recordsDF.createOrReplaceTempView("records");

 // Queries can then join DataFrames data with data stored in Hive.
 spark.sql("SELECT * FROM records r JOIN src s ON r.key = s.key").show();
 // +---+------+---+------+
 // |key| value|key| value|
 // +---+------+---+------+
 // |  2| val_2|  2| val_2|
 // |  2| val_2|  2| val_2|
 // |  4| val_4|  4| val_4|
 // ...

 spark.stop();
}
```

Core and SQL

22/04/2020 - Big Data