

# Structural Bioinformatics Training Workshop & Hackathon 2017

## Apache Spark Introduction

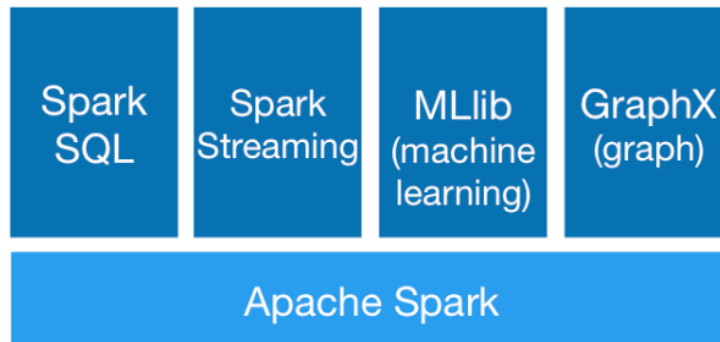
Yana Valasatava  
RCSB PDB

*Structural Bioinformatics Laboratory  
San Diego Supercomputer Center  
UC San Diego*

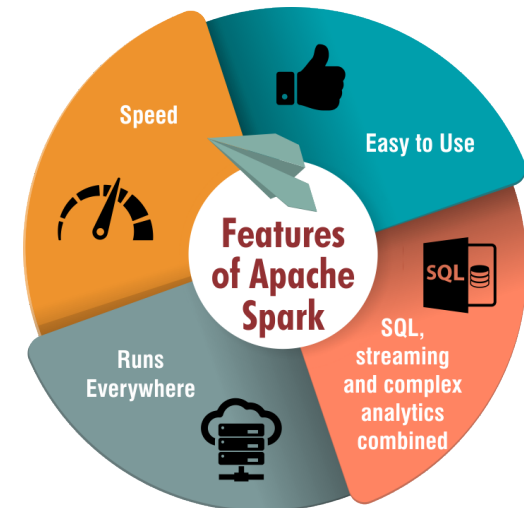
# Introduction to Apache Spark

Apache Spark is an open-source software framework that provides a distributed environment designed to store and process big data.

## *Apache Spark Ecosystem*



**Core API:** Python, Java, R, Scala



Spark offers support for multiple languages and makes it easy to build parallel applications.

# Initialize and close Spark

```
import org.apache.spark.SparkConf;  
import org.apache.spark.JavaSparkContext;
```

**SparkConf** object contains information about your application

```
SparkConf conf = new SparkConf()  
    .setAppName("myAppName")  
    .setMaster("local[*]")  
    .config("spark.driver.maxResultSize", "4g");
```

<https://spark.apache.org/docs/latest/configuration.html#available-properties>

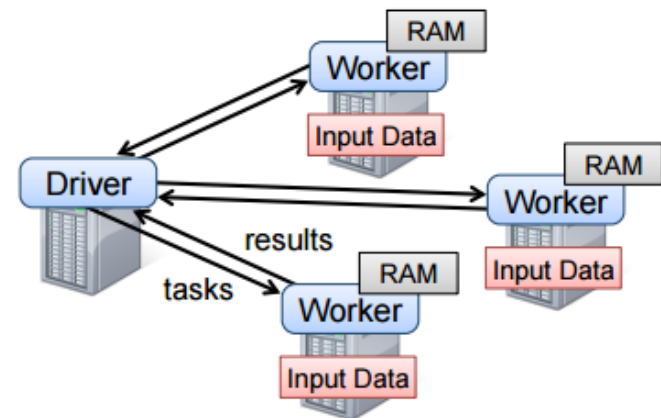
**SparkContext** is the entry point for interacting with Spark

```
JavaSparkContext sc = new JavaSparkContext(conf);  
...  
sc.stop();
```

# Distributed data structures

Spark revolves around the concept of a *resilient distributed dataset* (RDD):

- core Spark abstraction
- represents partitions across the cluster nodes
- enables parallel processing of datasets
- partitions can be in-memory or on-disk
- partitions can be recomputed on failure



There are two ways to create RDDs:

- parallelizing an existing collection in your driver program
- referencing a dataset in an external storage system

# Create JavaRDD

Parallelizing an existing Java collection:

```
List<String> data = Arrays.asList("pandas", "I like pandas");  
JavaRDD<String> lines = sc.parallelize(data, 10);
```

↑                    ↑  
*Java collection*    *Number of partitions*

❑ *Problem01*

Referencing to a datasets on external storage:

```
JavaRDD<String> lines = sc.textFile("data.txt");
```

❑ *Problem02*

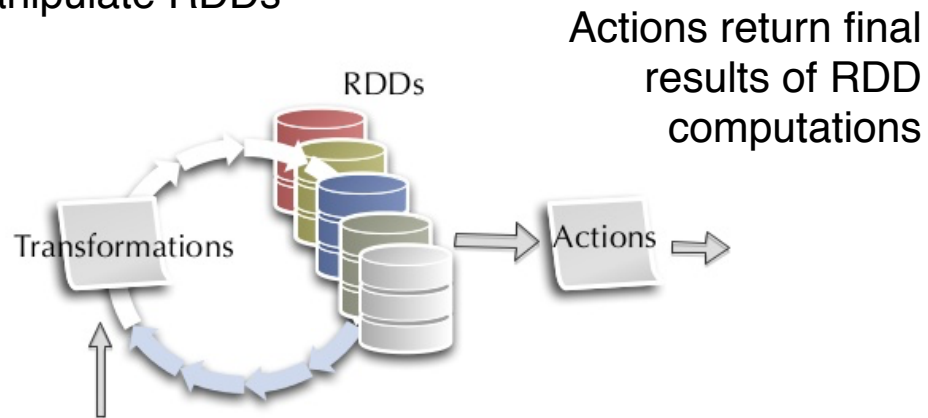
Apart from text files, Spark's Java API also supports several other data formats:

- `.wholeTextFiles` lets you read a directory containing multiple small text files;
- `.sequenceFile[K, V]` for Hadoop Sequence Files;
- other Hadoop input formats.

# RDD operations

Spark provides a rich set of operators to manipulate RDDs

Transformations	Actions
<code>map(func)</code>	<code>take(N)</code>
<code>flatMap(func)</code>	<code>count()</code>
<code>filter(func)</code>	<code>collect()</code>
<code>groupByKey()</code>	<code>reduce(func)</code>
<code>reduceByKey(func)</code>	<code>takeOrdered(N)</code>
<code>mapValues(func)</code>	<code>top(N)</code>
...	...



```
JavaRDD<String> data = sc.parallelize(Arrays.asList("Hello World!", "Hi"));
JavaRDD<String> twoWords = data.filter(func1);
twoWords.count();
twoWords.collect();
```

`.cache()`  
`.persist()`



# RDD transformations: filter()

Filter :

`filter(f : T  $\Rightarrow$  Bool)`

`JavaRDD<T>n  $\Rightarrow$  JavaRDD<T>m $\leq$ n`

Passing anonymous functions:

```
JavaRDD<Integer> rdd2 = rdd1.filter( e -> e % 2 == 0 )
```

□ Problem03

Passing functions:

```
JavaRDD<Integer> rdd2 = rdd1.filter(new Function<Integer, Boolean>() {  
    @Override  
    public Boolean call(Integer e) throws Exception {  
        return e % 2 == 0;  
    }  
});
```

the interfaces available in  
the *org.apache.spark.api.java.function* package

# RDD transformations: map() and flatMap()

Map :	$\text{map}(f : T \Rightarrow U)$	$\text{JavaRDD}\langle T \rangle_n \Rightarrow \text{JavaRDD}\langle U \rangle_n$
FlatMap :	$\text{flatMap}(f : T \Rightarrow \text{List}\langle U \rangle)$	$\text{JavaRDD}\langle T \rangle_n \Rightarrow \text{JavaRDD}\langle U \rangle_{m \geq n}$

```
JavaRDD<Integer> numbersRDD = context.parallelize(Arrays.asList(1,2,3));  
JavaRDD<Integer> squaresRDD = numbersRDD.map( n -> n*n );  
JavaRDD<String> stringRDD = numbersRDD.map( n -> String.valueOf(n));  
JavaRDD<Integer> multipliedRDD = numbersRDD  
    .flatMap( n->Arrays.asList(n,n*2,n*3).iterator());
```

□ Problem04



# RDD actions: reduce() and collect()

Spark RDD *reduce* function reduces the elements of this RDD using the specified commutative and associative binary operator

```
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(8,0,5,3,10,6));  
  
long total = rdd.reduce( (n1,n2) -> n1+n2 );
```

Return all the elements of the dataset as an array

```
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(8,0,5,3,10,6));  
  
List<Integer> list = rdd.collect();
```

# Working with Key/Value Pairs: JavaPairRDD

Key/Value pairs are stored using the *scala.Tuple2* class

Calling a function that returns a key/value pair, for instance, the `mapToPair ()`:

```
List<String> data = Arrays.asList("pandas", "I like pandas");  
JavaRDD<String> lines = sc.parallelize(data);  
JavaPairRDD<Integer, String> wordsCount = lines  
    .mapToPair(line -> new Tuple2(line.split(" ").length, line));
```

❑ *Problem05*

Some methods on *SparkContext* produce pair RDD by default for reading files in certain Hadoop formats

```
sc.sequenceFile(path, Text.class, BytesWritable.class);
```

# Working with JavaPairRDD

```
JavaPairRDD<Integer, String> wordsCount;  
JavaRDD<Integer> rddKeys = wordsCount.keys();  
JavaRDD<String> rddValues = wordsCount.values();
```

Pass each value in the pair RDD through a map function without changing the keys;

```
JavaPairRDD<K,V> rdd1;  
JavaPairRDD<K,U> rdd2 = rdd1.mapValues(Function<V, U> f);
```

❑ *Problem06, 07*

Pass each value in the pair RDD through a flatMap function without changing the keys

```
JavaPairRDD<K,V> rdd1;  
JavaPairRDD<K,U> rdd2 = rdd1.flatMapValues(Function<V,Iterable<U>> f);
```

❑ *Problem08*

# JavaPairRDD: flatMapToPair()

PairFlatMapFunction returns zero or more key-value pair records from each input record

```
JavaPairRDD<K,V> rdd1;  
JavaPairRDD<K,U> rdd2 = rdd1  
    .flatMapToPair(PairFlatMapFunction<T,K,V> f);  
                    ↑  
                    Tuple2  
                    ↑  
                    Iterator<Tuple2<K,V>> call(T t)
```

❑ *Problem09*

FlatMapFunction returns a record from each input key-value record

```
JavaPairRDD<K,V> rdd1;  
JavaRDD<U> rdd2 = rdd1.flatMap(FlatMapFunction<T,U> f);
```

# Transformations on PairRDDs

When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs

```
JavaPairRDD<String, String> rdd1;  
JavaPairRDD<String, Iterable<String>> rdd2 = rdd1.groupByKey();
```

❑ *Problem10*

When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*:  $(V, V) \Rightarrow V$

```
JavaPairRDD<String, Integer> rdd1;  
JavaPairRDD<String, Integer> rdd2 = rdd1.reduceByKey(func);
```

❑ *Problem11*

# Actions on PairRDDs

<b>collect()</b>	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
<b>count()</b>	Return the number of elements in the dataset.
<b>first()</b>	Return the first element of the dataset (similar to take(1)).
<b>take(<math>n</math>)</b>	Return an array with the first $n$ elements of the dataset.

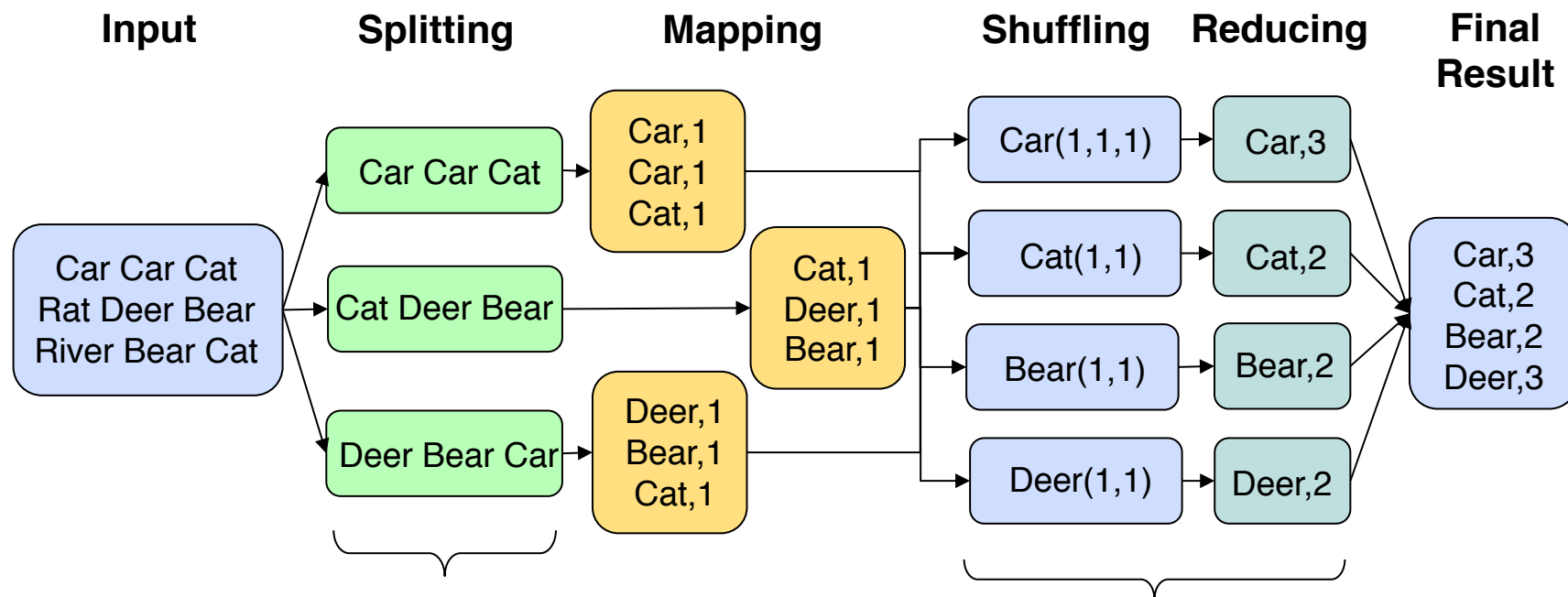
```
JavaPairRDD<String, Integer> pairRdd;  
  
List<Tuple2<String, Integer>> list = pairRdd.collect();
```

<b>saveAsTextFile(<math>path</math>)</b>	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem
<b>saveAsSequenceFile(<math>path</math>)</b>	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem

# Word count with Spark

The overall word count process:

```
JavaPairRDD<String, Integer> pairRdd = rdd.mapToPair();
```



```
JavaRDD<String> rdd = sc  
    .parallelize();  
pairRdd.reduceByKey((x, y) -> x + y)
```



# Apache Spark Dataset

The Datasets API provides the benefits of RDDs (strong typing, ability to use lambda functions) with the benefits of Spark SQL's optimized execution engine

```
SparkSession ss = SparkSession.builder()  
    .master("local[*]").appName("app")  
    .config("spark.driver.maxResultSize", "4g")  
    .config("spark.executor.memory", "4g")  
    .getOrCreate();
```

```
Dataset<Row> data = sparkSession.read().scv(...);
```

❑ *Problem12*

```
List<Person> data;  
Dataset<Row> dataset = ss.createDataset(data, Person.class);
```

❑ *Problem13*

# Querying Dataset: SQL API

SQL statements can be run by using the SQL methods provided by spark

```
Dataset<Row> peopleDF = spark.createDataFrame(peopleRDD, Person.class);

peopleDF.createOrReplaceTempView("people");

Dataset<Row> teenagersDF = spark
    .sql("SELECT name FROM people WHERE age BETWEEN 13 AND 19");
```

```
Dataset<Row> peopleDF;
Dataset<Row> gilrsDF = peopleDF.select(col("gender").equalTo("F"))
    .filter(col("age").geq(13)
        .and(col("age").geq(19)));
```

□ *Problem14*

# Questions?