Classe Scanner in java.utils
useDelimiter("regex") imposta l'espressione regolare in argomento come delimitatore.

hasNext()/hasNextInt()/ hasNextDouble()… Ritorna **true** se il token successivo può essere interpretato come String/int/double…

next()/nextInt()/nextDouble()…
Restituisce il token successivo interpretato come String/int/double...
Puossono lanciare
InputMismatchException
Se si cerca di interpretare in modo errato
NoSuchElementException se lo scanner non ha più token
IllegalStateException se lo scanner è stato chiuso

Classe String
contains(CharSequence s)
Returns true if and only if this string contains the specified sequence of char values.
split(String regex[, int limit])
Splits this string around matches of the given regular expression.
toLowerCase()/toUpperCase()
Converts all of the characters in this String to lower/upper case using the rules of the default locale.
trim()
Returns a copy of the string, with leading and trailing whitespace omitted.

ESPRESSIONI REGOLARI
ab: il pattern costituito da 2 caratteri, a e b.
[ab]: 1 carattere, a o b.
[^ab]: 1 carattere diverso da a e b.
[a-g]: 1 carattere compreso tra a e g compresi
d: una cifra; *D*: un carattere diverso da una cifra.
s: uno spazio (oppure tab, carriage return); *S*: un carattere diverso da uno spazio
\\w: una lettera o una cifra o "_"; *W*: un carattere diverso dai precedenti.
Quantificatori: * zero o più, + 1 o più, *?* zero o uno, *{n}* n volte, *{m, n}* da m a n volte.

java.util.Comparator
comparing(): confronta gli elementi in base alla chiave estratta applicando la funzione di mappatura fornita come primo argomento. È possibile specificare il comparator da usare, altrimenti è usato *naturalOrder();*
naturalOrder(): gli elementi sono confrontati in base all'ordinamento naturale specificato dalla classe.
reverseOrder(): ordine inverso.

Hadoop

Input formats:

TextInputFormat
An input format for plain text files
Broken into lines
Key: position of the line in the file (offset)
Value: content of the line

KeyValueTextInputFormat
An input format for plain text files in which each line has the format
key<separator>value
By default <separator> is \t
To change it:

```
conf.set("mapreduce.input.keyvaluelinerecordreader.key.value.separator", ",");
```

Key: text preceding the <separator>
Value: text following the <separator>

SequenceFileInputFormat
An input format for sequential/binary files

Output formats:

TextOutputFormat
An output format for plain text files
For each output (key, value) pair, one line in output with format
key\tvalue\n

SequenceOutputFormat
An output format for sequential/binary files

Data Types:

Text → String
IntWritable → Integer
LongWritable → Long
FloatWritable → Float
All implements *Writable* and *WritableComparable*
Keys must implement *WritableComparable*
Values must implement *Writable*

Mapper:

```
class MyMapper extends
Mapper<K1,V1,K2,V2> {
```

K1: input key type
V1: input value type
K2: output key type
V2: output value type

```
protected void map(K1 key, V1
value, Context context) throws
IOExcaption,
InterruptedException {
    context.write(new
outputKey, new outputValue);
}
}
```

Reducer:

```
class MyReducer extends
Reducer<K1,V1,K2,V2> {
```

K2: Input key type (output key type of Mapper)
V2: Input value type (output value type of Mapper)
K3: Output key type
V3: Output value type

```
protected void reduce(K2 key,
Iterable<V2> values, Context
context) throws IOException,
InterruptedException {
    context.write(new
outputKey, new outputValue);
}
}
```

Combiner:

Only if Reduce is commutative and associative!!
In the run method of the driver:

```
job.setCombinerClass()

class MyCombiner extends
Reducer<K1,V1,K2,V2> {
```

K2: Input and output key type (output key type of Mapper)
V2: Input and output value type (output value type of Mapper)

```
protected void reduce(K1 key,
Iterable<V1> values, Context
context) throws IOException,
InterruptedException {
    context.write(new
outputKey, new outputValue);
}
}
```

Sharing parameters:

In the driver:

```
conf.set(“property-name”,”val”);
```

In the Mapper or Reducer or …

```
context.getConfiguration().get(“property-name”);
```

returns a *String* containing the value of the property identified by *property-name*

Counters:

In the driver:

```
public static enum COUNTERS {
    COUNTER1,
    COUNTER2
}
job.getCounters().findCounter(COUNTERS.COUNTER1);
```

In the Mapper or Reducer or …

```
context.getCounter(countername).increment(value)
```

Personalized Data Types:

```
public class MyType implements
Writable[Comparable] {
private int field1;
private float field2;
…
@Override
public void write(DataOutput
out) throws IOException {
     out.writeInt(field1);
     out.writeFloat(field2);
}
@Override
public void
readFields(DataInput in) throws
IOException{
     field1 = in.readInt();
     field2 = in.readFloat();
}
[@Override
public int
compareTo(WritableComparable o)
{
     return
Float.compare(field2,
o.field2);
}
@Override
public int hashCode() {
     return
Float.floatToIntBits(field2);
}
]
}
```

Numerical summarization

Group elements by a key and compute numerical aggregates per group.

Mapper: emit (key, value) pairs, where:

- *key:* fields used to define groups
- *value:* fields used to compute stats

Reducer: receive the list of numerical values for each key (group) and compute the final stats

Combiners: can be used for speedup if the statistic is associative

e.g.: word count, record count, min, max, count, average, median, std-dev…

Counting with counters

Compute count summarizations of datasets

Mapper: process each input and increment a set of counters

Map-only job. The results are stored/printed by the driver

Map-only jobs

Some applications like record filtering…
Avoid reduce and shuffle-and-sort
The output of Mappers is stored in HDFS

- Implement the *map* method
- In the driver:
  `job.setNumReduceTasks(0)`

In-Mapper combiner

Mapper interface contains also a *setup* and a *cleanup* method that defaults to empty methods.

setup: is called only once for each Mapper before the calls to *map*. Can be used to setup the values of the in-mapper variables that are used to keep statistics and preserve the state (locally for each mapper) within and across calls to the map method

cleanup: is called only once for each mapper after the calls to *map*. Can be used to emit (key, value) pairs based on the values of in-mapper vars/stats.

To implement an in-Mapper combiner:

- Initialize in-mapper vars in the *setup*
- Update the vars in the *map* (usually *map* does not emit any pair)
- Emit (key, value) pairs in the *cleanup* method based on the values of the internal variables

In-Mapper combiners performs better than a proper Combiner, but can use a **limited amount of memory**.

Inverted index

Build an index from input data for faster searches or data enrichment

Mapper: emit (key, value) pairs where:

- *key:* fields to index (keywords)
- *value:* identifier of the objects to associate with each keyword

Reducer: concatenate identifiers

No combiners

e.g. Web search engine: word → URLs

Filtering

Filter input records that are not of interest.

Mapper: emit only the records that satisfy the filtering rule. Map-only job.

Distinct

Find a unique set of values/records

Mapper: emit each input record as key associated with a null value

Reducer: Emit one *(key, value)* pair for each input *(key, list-of-values)*

Top K

Select a small set of the most important records, according to a ranking function.
Mapper: Initialize an in-mapper top-K list and update it in the *map* method. The *cleanup* emits each record in the list as value associated with a null key.
Reducer: single reducer, merges the local lists emitted by the mappers

Distributed Cache

Efficiently shared read-only, little files
In the driver:

```
job.addCacheFile(path)
```

In the Mapper/Reducer:

```
URI[] urisCachedFiles =
context.getCacheFiles();
BufferedReader file = new
BufferedReader(new
FileReader(new
File(urisCachedFiles[0].getPath
()))));
…
file.close();
```

Binning

Organize input records into categories
Driver: use MultipleOutputs to set the list of bins (output files)
Mapper: For each input pair, select the output bin and emit the pair in that file. Map-only job.

Shuffling

Randomize the order of the records
Mapper: For each input record, emit a (key, value) pair

- *key:* a random number
- *value:* the input record

Reducer: emit one pair for each value of the input pair

Natural join

Join the content of two relations

Reduce side

One mapper for each table, emit one *(key, value)* pair for each input record:

- *key:* value of the common attributes
- *value:* concatenation of the table name and the content of the record

Reducer: iterate over the values with each key and compute the natural join for that key

Map side

Mapper: small table is in the distributed cache. Perform the natural join for each element of the large table with the small one. Map only job.
Same pattern can be used for the other SQL joins

Multiple inputs

Datasets are split in files with different formats → specify a different mapper for each dataset, that shall emit consistent (key, value) pairs.
In the Driver:
For each input path

```
MultipleInputs.addInputPath(jo
b, path,
TextInputFormat.class,
Mapper.class);
```

Multiple outputs

Store the output pairs in different files (e.g. useful in splitting operations) with a prefix that specify its content.
In the driver:
For each output file:

```
MultipleOutputs.addNamedOutput
(job, “type1”,
TextOutputFormat.class,
Text.class,
NullWritable.class);
```

Define a private variable in the Mapper of a map-only job or in the reducer

```
private MultipleOutputs<Text,
NullWritable> mos = null;
```

In the setup of the Mapper of a map-only job or in the setup of the Reducer,

```
mos = new
MultipleOutputs<Text,
NullWritable>(context);
```

To write in the file of interest:

```
mos.write(“type1”, key,
value);
```

In the cleanup,

```
mos.close()
```

```
protected void map(
     LongWritable  key,
     Text value,
     Context context) throws
IOException, InterruptedException {
     String[]
fields=value.toString().split(",");
      context.write(new
Text(fields[1]), new
FloatWritable(Float.parseFloat(fiel
ds[3])));
}

protected void reduce(
    Text key,
    Iterable<FloatWritable> values,
    Context context) throws
IOException, InterruptedException {
…
     context.write(…);
}
```

Spark

To access the spark functionalities, use the *SparkContext* object that is defined by the driver:

```
SparkConf conf = new SparkConf().setAppName("Spark application")
JavaSparkContext sc = new JavaSparkContext(conf);
```

Spark RDDs

`JavaRDD<T>` is an immutable distributed collection of objects.

Creation

Create a new *JavaRDD* from the lines of an input file:

```
JavaRDD<String> lines = sc.textFile(inputPath[, numPartitions]);
```

Create a new *JavaRDD* from a local Java collection:

```
JavaRDD<String> distList = sc.parallelize(inputList[, numPartitions]);
```

Trasformations

`filter(Function<T, Boolean> predicate);` returns a new RDD containing only the elements that satisfy the *predicate*

`map(Function<T,R> mapper);` returns a new RDD applying the *mapper* function on each element of the origin RDD.

`flatMap(FlatMapFunction<T,R> flatMapper);` returns a new RDD applying the provided mapper to each element of the origin RDD. The flatMapper is a function that takes an element of type T (type of the elements of the origin RDD) and returns an Iterator of elements of type R. The resulting RDD contains the concatenation of the iterators.

`distinct();` returns a new RDD containing the distinct elements of the input one, without duplicates.

`sample(boolean withReplacement, double fraction);` returns a new RDD containing a random sample of the elements from the origin RDD.

Tuples

`Tuple2<K,V>` represents a *(key, value)* pair.

Constructor: `new Tuple2(key, value);`

Getters: `._1()` retrieves the key,
`._2()` retrieves the vaue;

Set transformations

`union(JavaRDD<T> secondInputRDD);`
`intersection(JavaRDD<T> secondInputRDD);`
`subtract(JavaRDD<T> secondInputRDD);`
`cartesian(JavaRDD<T> secondInputRDD);`

Actions

`collect();` Returns a local Java *List* with the objects of the RDD.

`saveAsTextFile(outputPath);` **Save** an RDD in a text file on the HDFS.

`Count();` return the number of element in this RDD, as long.

`CountByValue();` Returns a local Java *Map* containing information about the number of times each element occurs in the RDD (a Map<T, Long>).

`Take(int n);` Returns a *List* with the first n elements of the RDD.

`First();` Returns the first element of the RDD.

`Top(int n);` Returns a *List* with the n largest elements in the RDD.

`TakeOrdered(int n, Comparator<T> comp);` returns a *List* with n smallest elements in the RDD.

`TakeSample(boolean withReplacement, int n[, long seed]);` Returns a *List* containing n random elements of the RDD.

`Reduce(Function2<T,T,T> combiner);` Returns a single object obtained combining the objects of the RDD using a user provided function that must be associative and commutative.

`Fold(T zero, Function2<T,T,T> combiner);` Return a single object obtained combining the objects of the RDD by using a user provided function that must be associative and a zero value.

`Aggregate(U zero, Function2<U, T, U> seqOp, Function2<U,U,U> combOp);` Return a single object obtained combining the objects of the RDD and an initial zero value by using two user provided functions that must be associative and commutative.

Spark PairRDDs

RDDs of (key, value) pairs
Standard RDD operations (T is Tuple2<K,V>) plus…

Creation

From regular RDDs using the `mapToPair(PairFunction<T,K2,V2> f)` transformation. f takes an element of the origin “regular” RDD and returns a Tuple2, with key and value of the new PairRDD.
From java collections, using the `parallelizePairs(List<Tuple2<K, V>>);` method of the SparkContext.

Transformations

`reduceByKey(Function2<V,V,V> f);` Create a new *PairRDD* where there is one pair for each distinct key of the input *pairRDD*. The value associated to the key is computed applying the *f* function <u>(associative and commutative)</u> on the values associated with that key in the input *pairRDD*.

`foldByKey(V zero, Function2<V,V,V> f);` Create a new *PairRDD* where there is one pair for each distinct key of the input *pairRDD*. The value associated to the key is computed using the *f* function <u>(associative)</u> on the values associated with that key in the input *pairRDD*, and a zero value.

`combineByKey(Function<V,U> createCombiner, Function2<U,V,U> mergeValue, Function2<U,U,U> mergeCombiner);` Create a new *PairRDD* where there is one pair for each distinct key of the input *pairRDD*. The value associated to the key is computed using *createCombiner* to create an initial value and *mergeValue* <u>(associative and commutative)</u> on the values associated with that key in the input *pairRDD*. The partial results are merged using the *mergeCombiner.*

`groupByKey();` Create a new *PairRDD* where there is one pair for each distinct key of the input PairRDD. The value associated to the key is the list of values associated with that in the input *PairRDD*.

`mapValues(Function<V,U> mapper);` apply the *mapper* function on the value of each pair of the input *PairRDD*.

`keys();` returns an RDD with the keys (not unique) of the input PairRDD.

`values();` returns an RDD with the values (not deduplicated) of the origin PairRDD.

`flatMapValues(Function<V, Iterable<U>> f);` apply *f* to the value of each pair of the input PairRDD. The function returns an iterable. The resulting PairRDD is obtained associating each key with each element of the iterable generated for that key.

`SortByKey();` Return a new PairRDD obtained by sorting in ascending order the pairs of the input PairRDD by the key. K must implement Ordered.

Trasformations on pairs of PairRDDs

`subtractByKey(JavaPairRDD<K,U> other);` Create a new *PairRDD* containing only the pairs of the input *PairRDD* associated with a key that is not a key of the *other PairRDD*.

`join(JavaPairRDD<K,U> other);` Returns a *PairRDD<K,Tuple2<V,U>>* in which each pair of the input *PairRDD* is combined with all the pairs of the *other* with the same key.

`cogroup(JavaPairRDD<K,U>);` Returns a *PairRDD<K, Tuple2<Iterable <V>, Iterable<U>>>* that associates to each key the list of values of the input *PairRDD* and the list of values of the *other*

Actions

`countByKey();` Returns a local *Map* that associates to each key the number of elements associated to that key in the *PairRDD*. <u>**Attention to the number of distinct keys!!**</u>

`collectAsMap();` Returns a local *Map* that associates each distinct key to one of the values associated to that key in the *PairRDD*.

`Lookup(K key);` Return a local *List* of values associated with the specified *key*

DoubleRDD

RDD of doubles but different from JavaRDD<Double>
Create with `mapToDouble(DoubleFunction<T> f);` or `flatMapToDouble(DoubleFlatMapFunction<T> f);` on standard RDDs.
Or with `parallelizeDoubles();` of the SparkContext object.

Actions

Provides the following actions with the obvious meaning:
`sum(), mean(), stdev(), variance(), max(), min().`

Accumulators

Shared variables that efficiently support parallelism. They can be used to implement counters.

A LongAccumulator can be defined in the driver calling `sc.sc().longAccumulator();`

Similarly, a *DoubleAccumulator* can be obtained with `sc.sc().doubleAccumulator();`

The value of an accumulator can be increased using the `add(value)` method of the AccumulatorV2 class and retrieved (only in the driver, not in the lambdas) with `value()`.

Broadcast variables

Read only (medium/large) shared variables.

Better then standard variables because are sent only once to the worker nodes that need it.

Created with `sc.broadcast(T value)` that returns an object of class Broadcast<T>.

The content is retrieved using the `value()` method on that object.

Cache

If an RDD must be used more that once, it could be more efficient to cache its content instead of computing it again. To do this, call

`persist(StorageLevel level);`

Where level can be one of:

```
StorageLevel.MEMORY_ONLY()
StorageLevel.MEMORY_AND_DISK()
StorageLevel.MEMORY_ONLY_SER()
StorageLevel.MEMORY_AND_DISK_SER()
StorageLevel.DISK_ONLY()
StorageLevel.NONE()
StorageLevel.OFF_HEAP()
StorageLevel.MEMORY_ONLY_2()
StorageLevel.MEMORY_AND_DISK_2()
StorageLevel.MEMORY_ONLY_SER_2()
StorageLevel.MEMORY_AND_DISK_SER_2()
```

`cache();` is equivalent to *persist*() with storage level MEMORY_ONLY.

`unpersist();` can be used to remove an RDD from the cache.

Spark SQL

Distributed SQL query engine for structured data processing. Usually more efficient than RDDs.

Spark SQL funtionalities are based on the *SparkSession* class. To obtain an object of that class, call `SparkSession ss = SparkSession.builder() .appName("App Name").getOrCreate();`

To close it use the `stop()` method on the object.

DataFrame

Distributed collection of data organized into named columns (like relational tables)

Creation

From csv files:

```
DataFrameReader dfr =
ss.read().format("csv").option("header", true)
.option("inferSchema", true);
Dataset<Row> df =
dfr.load(inputPath);
```

From "JSON Lines text format" files:

Like for csv, but `.format("json")`

For standard JSON files add `.option("multiline", true)`

<u>Reading a set of small JSON files is very slow</u>

Get an RDD<Row>:

`JavaRDD();`

Row

`FieldIndex(String columnName);`

Returns the index of the given field

`GetAs(String columnName);` Ritrieve the content of a field given its name.

`GetString(int position);`
`GetDouble(int position); ...`

Ritrieve the content of the field at the given position.

Dataset

More general than DataFrames.
Collections of objects that represents the structure of data.
Similar to RDD, but more efficient.
The objects must be JavaBean compliant:

- Implement *Serializable*
- All attributes must have public getters and setters
- All attributes should be private

Creation

From local collection:
`createDataset(List<T> data, Encoder<T> encoder);`
From DataFrame
`as(Encoder encoder);`
From csv or JSON:
Create a DataFrame and use as().
From RDD:
`createDataset(RDD<T> inRDD, Encoder<T> encoder);`

Operations

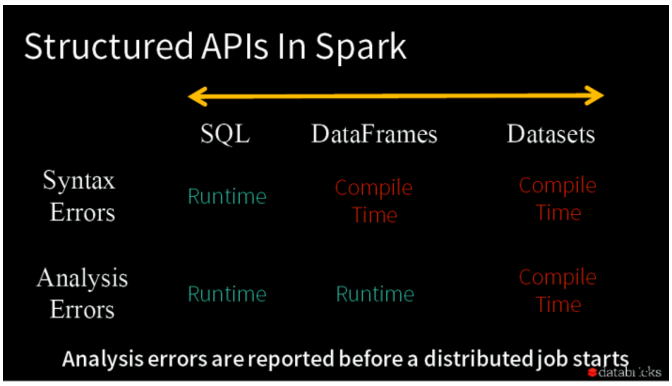
`show(int numRows);` Print on the stdout the first numRows rows of the Dataset.
`printSchema();` Prints on the stdout the schema of the Dataset
`count();` returns the number of rows of the Dataset as a long.
`select(String col1, …, String colN);` returns a new Dataset that contains only the specified columns.
`selectExpr(String expr1, …, String exprN);` Returns a DataFrame containing a set of columns comuted by combining the original columns.
`distinct();` returns a new dataset containing only the unique rows of the input Dataset.
`filter(String conditionExpr);`
`filter(FilterFunction<T> predicate);`
`where(String expression);`
Returns new Dataset that contains only the element that satisfy the predicate or the conditionExpr.
`map(Function1<T,U> mapper, Encoder<U> encoder);` Returns a new Dataset obtained applying the mapper to the elements of the input Dataset and encoding the returned object with the provided encoder
`flatMap()`
`join(Dataset<T> right, Column joinExprs);` joins two Datasets in a new DataFrame
`avg(column), count(column), num(column), abs(column), …`
Compute aggregates over the set of values of a column
`agg(aggregate functions);`
Returns a new DataFrame apllying multiple aggregate functions at once.
`groupBy(String col1, …, String colN);` split the input data in Groups. Returns a RelationalGroupedDataset. Can be used to caompute aggregate functions over groups.
`sort(String col1, …, String colN);`
`sort(Column col1, …, Column colN);` Returns a new dataset sorted in ascending order according to the specified columns. The second version can also sort in descending order applying the `desc()` method of the *Column*.

Save to disk

Convert to RDD and save the RDD to disk.
Use the `write()` method of Dataset combined with `format(String filetype)` and `save(String outputFolder)`.

SQL language

createOrReplaceTempView(String tableName); assign a table name to the DataFrame on which it is invoked.
sql(String sqlQueryText) of the SparkSession can be used to execute SQL-like queries.
Some SQL features are not supported (yet) (e.g. nested subqueries in the WHERE clause).



Spark streaming

API very similar to RDD, works on minibatches

A Spark streaming application runs until a specified timeout expires or until it gets killed. If the timeout is not specified, it will run forever until killed.

Access to the streaming API is provided through the spark streaming context:

`JavaStreamingContext jssc = new JavaStreamingContext(conf, Durations.seconds(10));`

The second parameter specify the duration of each batch.

To start the application, use the `start()` method of the *SparkStreamingContext* and to specify how long the application will run, `awaitTerminationOrTimeout([long milliseconds])`.

DStream

Sequence of RDDs

Creation

From a TCP socket:
socketTextStream(String hostname, int portNumber);
From an input HDFS folder:
textFileStream(String folder);
Lots of other sources

Transformations

`map(mapper)`, `flatMap(flatMapper)`, `filter(predicate)` `reduce(combiner)`, `count()`, `union(other)`, `join(other)`, `cogroup(other)`.
As in *RDDs*

`CountByValue();` Returns a new *PairDStream<T,Long>* that associate to each distinct elements its frequency.

Transform(func); Returns a new DStream obtained applying an RDD-to-RDD function to every RDD in the source Dstream.

Output operations (actions)

Print(); Prints the first 10 elements of every batch in the Dstream.

dstream().saveAsTextFile(prefix[, suffix]); Save the content of the Dstream on a text files, one folder for each batch. Name of the folder is prefix-TIME_IN_MS[.suffix].

Stateful operation

updateStateByKey(func); transformation: func takes a state varable of type Optional<T> and an iterable of new values and returns an optional with the new state.

PairDStream

Sequence of PairRDDs

Creation

Transformations

The DStream transformations, plus reduceByKey(combiner); Returns a new PairDStream where the values of each key are aggregated using the provided combiner.

`transformToPair(func);` Returns a new PairDStream obtained applying a PairRDD-to-PairRDD function to every PairRDD in the source PairDstream.

Output Operations (actions)

The DStream output operations

Window operation

Apply transformations over a sliding window of data.

Creation

window(windowLength, slideInterval); Returns a new Dstream computed based on windowed batches of the source DStream. WindowLength is the duration of a window in number of batches, slidingInterval is the interval at which the window operation is performed, in number of batches.

Transformations

`countByWindow(windowLength, slideInterval);` Returns a new single element stream containing the number of elements of each window.

`reduceByWindow(combiner, windowLength, slideInterval);` Returns a new single-element stream, created by aggregating elements in the stream over a sliding interval using func (associative)

`countByValueAndWindow(windowLength, slideInterval);` Returns a new PairDStream that associate to each distinct element of the input DStream, the number of occurrencies over a sliding window.

`reduceByKeyAndWindow(combiner, windowLength, slideInterval);` Applied to PairDStream, returns a new PairDStream where values of each key are aggregated using the provided combiner.

Checkpoints

Checkpointing is useful for resiliency and necessary for some window and stateful transformations. It is enabled using `checkpoint(String folder)` method of the SparkStreamingContext.