pysparkling Documentation

Release 0.3.22

Sven Kreiss

June 18, 2016

Contents

Pysparkling provides a faster, more responsive way to develop programs for PySpark. It enables code intended for Spark applications to execute entirely in Python, without incurring the overhead of initializing and passing data through the JVM and Hadoop. The focus is on having a lightweight and fast implementation for small datasets at the expense of some data resilience features and some parallel processing features.

How does it work? To switch execution of a script from PySpark to pysparkling, have the code initialize a pysparkling Context instead of a SparkContext, and use the pysparkling Context to set up your RDDs. The beauty is you don't have to change a single line of code after the Context initialization, because pysparkling's API is (almost) exactly the same as PySpark's. Since it's so easy to switch between PySpark and pysparkling, you can choose the right tool for your use case.

When would I use it? Say you are writing a Spark application because you need robust computation on huge datasets, but you also want the same application to provide fast answers on a small dataset. You're finding Spark is not responsive enough for your needs, but you don't want to rewrite an entire separate application for the *small-answers-fast* problem. You'd rather reuse your Spark code but somehow get it to run fast. Pysparkling bypasses the stuff that causes Spark's long startup times and less responsive feel.

Here are a few areas where pysparkling excels:

- Small to medium-scale exploratory data analysis
- Application prototyping
- Low-latency web deployments
- Unit tests

Example: you have a pipeline that processes 100k input documents and converts them to normalized features. They are used to train a local scikit-learn classifier. The preprocessing is perfect for a full Spark task. Now, you want to use this trained classifier in an API endpoint. Assume you need the same pre-processing pipeline for a single document per API call. This does not have to be done in parallel, but there should be only a small overhead in initialization and preferably no dependency on the JVM. This is what `pysparkling` is for. Links: Documentation, Github, Issue Tracker

CHAPTER 1

Install

```
pip install pysparkling
```

or to install with all dependencies:

```
pip install pysparkling[hdfs,tests]
```

CHAPTER 2

Features

- Supports multiple URI scheme: `s3://`, `hdfs://`, `http://` and `file://`. Specify multiple files separated by comma. Resolves `*` and `?` wildcards.
- Handles `.gz`, `.zip`, `.lzma`, `.xz`, `.bz2`, `.tar`, `.tar.gz` and `.tar.bz2` compressed files. Supports reading of `.7z` files.
- Parallelization via `multiprocessing.Pool`, `concurrent.futures.ThreadPoolExecutor` or any other Pool-like objects that have a `map(func, iterable)` method.
- Plain pysparkling does not have any dependencies (use `pip install pysparkling`). Some file access methods have optional dependencies: `boto` for AWS S3, `requests` for http, `hdfs` for hdfs

CHAPTER 3

Examples

Some demos are in the notebooks docs/demo.ipynb and docs/iris.ipynb .

Word Count

```
from pysparkling import Context

counts = Context().textFile(
    'README.rst'
).map(
    lambda line: ''.join(ch if ch.isalnum() else ' ' for ch in line)
).flatMap(
    lambda line: line.split(' ')
).map(
    lambda word: (word, 1)
).reduceByKey(
    lambda a, b: a + b
)
print(counts.collect())
```

which prints a long list of pairs of words and their counts.

CHAPTER 4

Contents

4.1 API

A usual `pysparkling` session starts with either parallelizing a `list` or by reading data from a file using the methods `Context.parallelize(my_list)` or `Context.textFile("path/to/textfile.txt")`. These two methods return an `RDD` which can then be processed with the methods below.

4.1.1 RDD

class `pysparkling.`**`RDD`**(*partitions*, *ctx*)

In Spark's original form, RDDs are Resilient, Distributed Datasets. This class reimplements the same interface with the goal of being fast on small data at the cost of being resilient and distributed.

Parameters

- **`partitions`** (*list*) – A list of instances of `Partition`.
- **`ctx`** (Context) – An instance of the applicable `Context`.

`aggregate`(*zeroValue*, *seqOp*, *combOp*)

[distributed]

Parameters

- **`zeroValue`** – The initial value to an aggregation, for example `0` or `0.0` for aggregating `int` s and `float` s, but any Python object is possible. Can be `None`.
- **`seqOp`** – A reference to a function that combines the current state with a new value. In the first iteration, the current state is zeroValue.
- **`combOp`** – A reference to a function that combines outputs of seqOp. In the first iteration, the current state is zeroValue.

Returns Output of `combOp` operations.

Example:

```
>>> from pysparkling import Context
>>> seqOp = (lambda x, y: (x[0] + y, x[1] + 1))
>>> combOp = (lambda x, y: (x[0] + y[0], x[1] + y[1]))
>>> Context().parallelize(
...     [1, 2, 3, 4], 2
... ).aggregate((0, 0), seqOp, combOp)
(10, 4)
```

aggregateByKey(*zeroValue*, *seqFunc*, *combFunc*, *numPartitions=None*)
aggregate by key

Parameters

- **zeroValue** – The initial value to an aggregation, for example `0` or `0.0` for aggregating `int` s and `float` s, but any Python object is possible. Can be `None`.
- **seqFunc** – A reference to a function that combines the current state with a new value. In the first iteration, the current state is zeroValue.
- **combFunc** – A reference to a function that combines outputs of seqFunc. In the first iteration, the current state is zeroValue.
- **numPartitions** (*int*) – (optional) Not used.

Returns An RDD with the output of `combOp` operations.

Return type *RDD*

Example:

```
>>> from pysparkling import Context
>>> seqOp = (lambda x, y: x + y)
>>> combOp = (lambda x, y: x + y)
>>> r = Context().parallelize(
...     [('a', 1), ('b', 2), ('a', 3), ('c', 4)]
... ).aggregateByKey(0, seqOp, combOp).collectAsMap()
>>> (r['a'], r['b'])
(4, 2)
```

cache()
Once a partition is computed, cache the result.

Alias for `RDD.persist()`.

Example:

```
>>> from pysparkling import Context
>>> from pysparkling import CacheManager
>>>
>>> n_exec = 0
>>>
>>> def _map(e):
...     global n_exec
...     n_exec += 1
...     return e*e
>>>
>>> my_rdd = Context().parallelize([1, 2, 3, 4], 2)
>>> my_rdd = my_rdd.map(_map).cache()
>>>
>>> logging.info('no exec until here')
>>> f = my_rdd.first()
>>> logging.info('available caches in {1}: {0}'.format(
...     CacheManager.singleton().stored_idents(),
...     CacheManager.singleton(),
... ))
>>>
>>> logging.info('executed map on first partition only so far')
>>> a = my_rdd.collect()
>>> logging.info('available caches in {1}: {0}'.format(
...     CacheManager.singleton().stored_idents(),
```

```
...     CacheManager.singleton(),
... ))
>>>
>>> logging.info('now _map() was executed on all partitions and should'
...              'not be executed again')
>>> logging.info('available caches in {1}: {0}'.format(
...     CacheManager.singleton().stored_idents(),
...     CacheManager.singleton(),
... ))
>>> (my_rdd.collect(), n_exec)
([1, 4, 9, 16], 4)
```

cartesian (*other*)

cartesian product of this RDD with `other`

Parameters **other** (RDD) – Another RDD.

Return type *RDD*

Note: This is currently implemented as a local operation requiring all data to be pulled on one machine.

Example:

```
>>> from pysparkling import Context
>>> rdd = Context().parallelize([1, 2])
>>> sorted(rdd.cartesian(rdd).collect())
[(1, 1), (1, 2), (2, 1), (2, 2)]
```

coalesce (*numPartitions*, *shuffle=False*)

Parameters

- **numPartitions** (*int*) – Number of partitions in the resulting RDD.
- **shuffle** – (optional) Not used.

Return type *RDD*

Note: This is currently implemented as a local operation requiring all data to be pulled on one machine.

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize([1, 2, 3], 2).coalesce(1).getNumPartitions()
1
```

cogroup (*other*, *numPartitions=None*)

Groups keys from both RDDs together. Values are nested iterators.

Parameters

- **other** (RDD) – The other RDD.
- **numPartitions** (*int*) – Number of partitions in the resulting RDD.

Return type *RDD*

Example:

```
>>> from pysparkling import Context
>>> c = Context()
>>> a = c.parallelize([('house', 1), ('tree', 2)])
```

```
>>> b = c.parallelize([('house', 3)])
>>>
>>> [(k, sorted(list([list(vv) for vv in v])))
...  for k, v in sorted(a.cogroup(b).collect())
... ]
[('house', [[1], [3]]), ('tree', [[], [2]])]
```

collect()
returns the entire dataset as a list

Return type list

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize([1, 2, 3]).collect()
[1, 2, 3]
```

collectAsMap()
returns a dictionary for a pair dataset

Return type dict

Example:

```
>>> from pysparkling import Context
>>> d = Context().parallelize([('a', 1), ('b', 2)]).collectAsMap()
>>> (d['a'], d['b'])
(1, 2)
```

compute(*split*, *task_context*)
interface to extend behavior for specific cases

Parameters **split** (*Partition*) – a partition

count()
number of entries in this dataset

Return type int

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize([1, 2, 3], 2).count()
3
```

countApprox()
same as `RDD.count()`

Return type int

countByKey()
returns a `dict` containing the count for every key

Return type dict

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize(
...     [('a', 1), ('b', 2), ('b', 2)]
... ).countByKey()['b']
2
```

countByValue()
returns a `dict` containing the count for every value

Return type dict

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize([1, 2, 2, 4, 1]).countByValue()[2]
2
```

distinct(*numPartitions=None*)
returns only distinct elements

Parameters **numPartitions** (*int*) – Number of partitions in the resulting RDD.

Return type *RDD*

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize([1, 2, 2, 4, 1]).distinct().count()
3
```

filter(*f*)

Parameters **f** – A function that if it evaluates to true when applied to an element in the dataset, the element is kept.

Return type *RDD*

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize(
...     [1, 2, 2, 4, 1, 3, 5, 9], 3,
... ).filter(lambda x: x % 2 == 0).collect()
[2, 2, 4]
```

first()
returns the first element in the dataset

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize([1, 2, 2, 4, 1, 3, 5, 9], 3).first()
1
```

Works also with empty partitions:

```
>>> from pysparkling import Context
>>> Context().parallelize([1, 2], 20).first()
1
```

flatMap(*f*, *preservesPartitioning=True*)
map followed by flatten

Parameters

- **f** – The map function.
- **preservesPartitioning** – (optional) Preserve the partitioning of the original RDD. Default True.

Return type *RDD*

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize(['hello', 'world']).flatMap(
...     lambda x: [ord(ch) for ch in x]
... ).collect()
[104, 101, 108, 108, 111, 119, 111, 114, 108, 100]
```

flatMapValues (*f*)

map operation on values in a (key, value) pair followed by a flatten

Parameters **f** – The map function.

Return type *RDD*

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize([(1, 'hi'), (2, 'world')]).flatMapValues(
...     lambda x: [ord(ch) for ch in x]
... ).collect()
[(1, 104), (1, 105), (2, 119), (2, 111), (2, 114), (2, 108), (2, 100)]
```

fold (*zeroValue*, *op*)

Parameters

- **zeroValue** – The inital value, for example `0` or `0.0`.
- **op** – The reduce operation.

Returns The folded (or aggregated) value.

Example:

```
>>> from pysparkling import Context
>>> my_rdd = Context().parallelize([4, 7, 2])
>>> my_rdd.fold(0, lambda a, b: a+b)
13
```

foldByKey (*zeroValue*, *op*)

Fold (or aggregate) value by key.

Parameters

- **zeroValue** – The inital value, for example `0` or `0.0`.
- **op** – The reduce operation.

Return type *RDD*

Example:

```
>>> from pysparkling import Context
>>> my_rdd = Context().parallelize([('a', 4), ('b', 7), ('a', 2)])
>>> my_rdd.foldByKey(0, lambda a, b: a+b).collectAsMap()['a']
6
```

foreach (*f*)

applies `f` to every element

It does not return a new RDD like `RDD.map()`.

Parameters **f** – Apply a function to every element.

Return type None

Example:

```
>>> from pysparkling import Context
>>> my_rdd = Context().parallelize([1, 2, 3])
>>> a = []
>>> my_rdd.foreach(lambda x: a.append(x))
>>> len(a)
3
```

foreachPartition(*f*)
applies `f` to every partition

It does not return a new RDD like `RDD.mapPartitions()`.

Parameters **f** – Apply a function to every partition.

Return type None

fullOuterJoin(*other*, *numPartitions=None*)
returns the full outer join of two RDDs

The output contains all keys from both input RDDs, with missing keys replaced with None.

Parameters

- **other** (RDD) – The RDD to join to this one.
- **numPartitions** (*int*) – Number of partitions in the resulting RDD.

Return type *RDD*

Note: Creating the new RDD is currently implemented as a local operation.

Example:

```
>>> from pysparkling import Context
>>> sc = Context()
>>> rdd1 = sc.parallelize([('a', 0), ('b', 1)])
>>> rdd2 = sc.parallelize([('b', 2), ('c', 3)])
>>> sorted(
...     rdd1.fullOuterJoin(rdd2).collect()
... )
[('a', (0, None)), ('b', (1, 2)), ('c', (None, 3))]
```

getNumPartitions()
returns the number of partitions

Return type int

getPartitions()
returns the partitions of this RDD

groupBy(*f*, *numPartitions=None*)
group by f

Parameters

- **f** – Function returning a key given an element of the dataset.
- **numPartitions** (*int*) – Number of partitions in the resulting RDD.

Return type *RDD*

Note: Creating the new RDD is currently implemented as a local operation.

Example:

```
>>> from pysparkling import Context
>>> my_rdd = Context().parallelize([4, 7, 2])
>>> my_rdd.groupBy(lambda x: x % 2).mapValues(sorted).collect()
[(0, [2, 4]), (1, [7])]
```

groupByKey(*numPartitions=None*)
group by key

Parameters **numPartitions** (*int*) – Number of partitions in the resulting RDD.

Return type *RDD*

Note: Creating the new RDD is currently implemented as a local operation.

histogram(*buckets*)

Parameters **buckets** – A list of bucket boundaries or an int for the number of buckets.

Returns A tuple (bucket_boundaries, histogram_values) where bucket_boundaries is a list of length n+1 boundaries and histogram_values is a list of length n with the values of each bucket.

Example:

```
>>> from pysparkling import Context
>>> my_rdd = Context().parallelize([0, 4, 7, 4, 10])
>>> b, h = my_rdd.histogram(10)
>>> h
[1, 0, 0, 0, 2, 0, 0, 1, 0, 0, 1]
```

intersection(*other*)
intersection of this and other RDD

Parameters **other** (RDD) – The other dataset to do the intersection with.

Return type *RDD*

Note: Creating the new RDD is currently implemented as a local operation.

Example:

```
>>> from pysparkling import Context
>>> rdd1 = Context().parallelize([0, 4, 7, 4, 10])
>>> rdd2 = Context().parallelize([3, 4, 7, 4, 5])
>>> rdd1.intersection(rdd2).collect()
[4, 7]
```

join(*other*, *numPartitions=None*)

Parameters

- **other** (RDD) – The other RDD.
- **numPartitions** (*int*) – Number of partitions in the resulting RDD.

Return type *RDD*

Note: Creating the new RDD is currently implemented as a local operation.

Example:

```
>>> from pysparkling import Context
>>> rdd1 = Context().parallelize([(0, 1), (1, 1)])
>>> rdd2 = Context().parallelize([(2, 1), (1, 3)])
>>> rdd1.join(rdd2).collect()
[(1, (1, 3))]
```

keyBy(*f*)
key by f

Parameters **f** – Function that returns a key from a dataset element.

Return type *RDD*

Example:

```
>>> from pysparkling import Context
>>> rdd = Context().parallelize([0, 4, 7, 4, 10])
>>> rdd.keyBy(lambda x: x % 2).collect()
[(0, 0), (0, 4), (1, 7), (0, 4), (0, 10)]
```

keys()
keys of a pair dataset

Return type *RDD*

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize([(0, 1), (1, 1)]).keys().collect()
[0, 1]
```

leftOuterJoin(*other*, *numPartitions=None*)
left outer join

Parameters

- **other** (RDD) – The other RDD.
- **numPartitions** (*int*) – Number of partitions in the resulting RDD.

Return type *RDD*

Note: Creating the new RDD is currently implemented as a local operation.

Example:

```
>>> from pysparkling import Context
>>> rdd1 = Context().parallelize([(0, 1), (1, 1)])
>>> rdd2 = Context().parallelize([(2, 1), (1, 3)])
>>> rdd1.leftOuterJoin(rdd2).collect()
[(0, (1, None)), (1, (1, 3))]
```

lookup(*key*)
Return all the (key, value) pairs where the given key matches.

Parameters **key** – The key to lookup.

Return type list

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize([(0, 1), (1, 1), (1, 3)]).lookup(1)
[1, 3]
```

map(*f*)

Parameters **f** – map function for elements

Return type *RDD*

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize([1, 2, 3]).map(lambda x: x+1).collect()
[2, 3, 4]
```

mapPartitions(*f*, *preservesPartitioning=False*)

map partitions

Parameters **f** – map function for partitions

Return type *RDD*

Example:

```
>>> from pysparkling import Context
>>> rdd = Context().parallelize([1, 2, 3, 4], 2)
>>> def f(iterator):
...     yield sum(iterator)
>>> rdd.mapPartitions(f).collect()
[3, 7]
```

mapPartitionsWithIndex(*f*, *preservesPartitioning=False*)

map partitions with index

Parameters **f** – map function for (index, partition)

Return type *RDD*

Example:

```
>>> from pysparkling import Context
>>> rdd = Context().parallelize([9, 8, 7, 6, 5, 4], 3)
>>> def f(splitIndex, iterator):
...     yield splitIndex
>>> rdd.mapPartitionsWithIndex(f).sum()
3
```

mapValues(*f*)

map values in a pair dataset

Parameters **f** – map function for values

Return type *RDD*

max()

returns the maximum element

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize([1, 2, 3, 4, 3, 2], 2).max() == 4
True
```

mean()
returns the mean of this dataset

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize([0, 4, 7, 4, 10]).mean()
5.0
```

min()
returns the minimum element

name()
returns the name of the dataset

persist(*storageLevel=None*)
Cache the results of computed partitions.

Parameters **storageLevel** – Not used.

pipe(*command*, *env=None*)
Run a command with the elements in the dataset as argument.

Parameters

- **command** – Command line command to run.
- **env** (*dict*) – environment variables

Return type *RDD*

Warning: Unsafe for untrusted data.

Example:

```
>>> from pysparkling import Context
>>> piped = Context().parallelize(['0', 'hello', 'world']).pipe('echo')
>>> b'hello\n' in piped.collect()
True
```

randomSplit(*weights*, *seed=None*)
Split the RDD into a few RDDs according to the given weights.

Parameters

- **weights** – Determines the relative lengths of the resulting RDDs.
- **seed** (*int*) – Seed for random number generator.

Returns A list of RDDs.

Return type list

Note: Creating the new RDDs is currently implemented as a local operation.

Example:

```
>>> from pysparkling import Context
>>> rdd = Context().parallelize(range(500))
>>> rdd1, rdd2 = rdd.randomSplit([2, 3], seed=42)
>>> (rdd1.count(), rdd2.count())
(199, 301)
```

reduce(*f*)

Parameters **f** – A commutative and associative binary operator.

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize([0, 4, 7, 4, 10]).reduce(lambda a, b: a+b)
25
```

reduceByKey(*f*)
reduce by key

Parameters **f** – A commutative and associative binary operator.

Return type *RDD*

Note: This operation includes a *pysparkling.RDD.groupByKey()* which is a local operation.

Example:

```
>>> from pysparkling import Context
>>> rdd = Context().parallelize([(0, 1), (1, 1), (1, 3)])
>>> rdd.reduceByKey(lambda a, b: a+b).collect()
[(0, 1), (1, 4)]
```

repartition(*numPartitions*)

Parameters **numPartitions** (*int*) – Number of partitions in new RDD.

Return type *RDD*

Note: Creating the new RDD is currently implemented as a local operation.

rightOuterJoin(*other*, *numPartitions=None*)
right outer join

Parameters

- **other** (RDD) – The other RDD.
- **numPartitions** (*int*) – Number of partitions in new RDD.

Return type *RDD*

Note: Creating the new RDD is currently implemented as a local operation.

Example:

```
>>> from pysparkling import Context
>>> rdd1 = Context().parallelize([(0, 1), (1, 1)])
>>> rdd2 = Context().parallelize([(2, 1), (1, 3)])
>>> sorted(rdd1.rightOuterJoin(rdd2).collect())
[(1, (1, 3)), (2, (None, 1))]
```

sample(*withReplacement*, *fraction*, *seed=None*)
randomly sample

Parameters

- **withReplacement** – Not used.
- **fraction** – Specifies the probability that an element is sampled.

- **seed** – (optional) Seed for random number generator.

Return type *RDD*

Example:

```
>>> from pysparkling import Context
>>> rdd = Context().parallelize(range(100))
>>> sampled = rdd.sample(False, 0.1, seed=5)
>>> all(s1 == s2 for s1, s2 in zip(sampled.collect(),
...                                sampled.collect()))
True
```

sampleByKey(*withReplacement*, *fractions*, *seed=None*)
randomly sample by key

Parameters

- **withReplacement** – Not used.
- **fractions** – Specifies the probability that an element is sampled per Key.
- **seed** – (optional) Seed for random number generator.

Return type *RDD*

Example:

```
>>> from pysparkling import Context
>>> sc = Context()
>>> fractions = {"a": 0.2, "b": 0.1}
>>> rdd = sc.parallelize(
...     fractions.keys()
... ).cartesian(
...     sc.parallelize(range(0, 1000))
... )
>>> sample = dict(
...     rdd.sampleByKey(False, fractions, 2).groupByKey().collect()
... )
>>> 100 < len(sample["a"]) < 300 and 50 < len(sample["b"]) < 150
True
>>> max(sample["a"]) <= 999 and min(sample["a"]) >= 0
True
>>> max(sample["b"]) <= 999 and min(sample["b"]) >= 0
True
```

sampleStdev()
sample standard deviation

Return type float

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize([1, 2, 3]).sampleStdev()
1.0
```

sampleVariance()
sample variance

Return type float

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize([1, 2, 3]).sampleVariance()
1.0
```

saveAsPickleFile(*path*, *batchSize=10*)
save as pickle file

Returns `self`

Return type *RDD*

> **Warning:** The output of this function is incompatible with the PySpark output as there is no pure Python way to write Sequence files.

Example:

```
>>> from pysparkling import Context
>>> from tempfile import NamedTemporaryFile
>>> tmpFile = NamedTemporaryFile(delete=True)
>>> tmpFile.close()
>>> d = ['hello', 'world', 1, 2]
>>> rdd = Context().parallelize(d).saveAsPickleFile(tmpFile.name)
>>> 'hello' in Context().pickleFile(tmpFile.name).collect()
True
```

saveAsTextFile(*path*, *compressionCodecClass=None*)
save as text file

If the RDD has many partitions, the contents will be stored directly in the given path. If the RDD has more partitions, the data of the partitions are stored in individual files under `path/part-00000` and so on and once all partitions are written, the file `path/_SUCCESS` is written last.

Parameters

- **path** – Destination of the text file.
- **compressionCodecClass** – Not used.

Returns `self`

Return type *RDD*

sortBy(*keyfunc*, *ascending=True*, *numPartitions=None*)
sort by keyfunc

Parameters

- **keyfunc** – Returns the value that will be sorted.
- **ascending** – Default is True.
- **numPartitions** (*int*) – Default is None. None means the output will have the same number of partitions as the input.

Return type *RDD*

Note: Sorting is currently implemented as a local operation.

Examples:

```
>>> from pysparkling import Context
>>> rdd = Context().parallelize([5, 1, 2, 3])
>>> rdd.sortBy(lambda x: x).collect()
[1, 2, 3, 5]
```

```
>>> from pysparkling import Context
>>> rdd = Context().parallelize([1, 5, 2, 3])
>>> rdd.sortBy(lambda x: x, ascending=False).collect()
[5, 3, 2, 1]
```

sortByKey(*ascending=True*, *numPartitions=None*, *keyfunc=<operator.itemgetter object>*)

sort by key

Parameters

- **ascending** – Default is True.
- **numPartitions** (*int*) – Default is None. None means the output will have the same number of partitions as the input.
- **keyfunc** – Returns the value that will be sorted.

Return type *RDD*

Note: Sorting is currently implemented as a local operation.

Examples:

```
>>> from pysparkling import Context
>>> rdd = Context().parallelize(
...     [(5, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
... )
>>> rdd.sortByKey().collect()[0][1] == 'b'
True
```

```
>>> from pysparkling import Context
>>> rdd = Context().parallelize(
...     [(1, 'b'), (5, 'a'), (2, 'c'), (3, 'd')]
... )
>>> rdd.sortByKey(ascending=False).collect()[0][1] == 'a'
True
```

stats()

Return type *StatCounter*

Example:

```
>>> from pysparkling import Context
>>> d = [1, 4, 9, 16, 25, 36]
>>> s = Context().parallelize(d, 3).stats()
>>> sum(d)/len(d) == s.mean()
True
```

stdev()

standard deviation

Return type float

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize([1.5, 2.5]).stdev()
0.5
```

subtract(*other*, *numPartitions=None*)

Parameters

- **other** (RDD) – The RDD to subtract from the current RDD.
- **numPartitions** (*int*) – Currently not used. Partitions are preserved.

Return type *RDD*

Example:

```
>>> from pysparkling import Context
>>> rdd1 = Context().parallelize([(0, 1), (1, 1)])
>>> rdd2 = Context().parallelize([(1, 1), (1, 3)])
>>> rdd1.subtract(rdd2).collect()
[(0, 1)]
```

sum()

sum of all the elements

Return type float

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize([0, 4, 7, 4, 10]).sum()
25
```

take(*n*)

take n elements and return them in a list

Only evaluates the partitions that are necessary to return n elements.

Parameters **n** (*int*) – Number of elements to return.

Return type list

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize([4, 7, 2]).take(2)
[4, 7]
```

Another example where only the first two partitions only are computed (check the debug logs):

```
>>> from pysparkling import Context
>>> Context().parallelize([4, 7, 2], 3).take(2)
[4, 7]
```

takeSample(*n*)

take sample

Assumes samples are evenly distributed between partitions.

Only evaluates the partitions that are necessary to return n elements.

Parameters **n** (*int*) – The number of elements to sample.

Return type list

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize([4, 7, 2]).takeSample(1)[0] in [4, 7, 2]
True
```

Another example where only one partition is computed (check the debug logs):

```
>>> from pysparkling import Context
>>> d = [4, 9, 7, 3, 2, 5]
>>> Context().parallelize(d, 3).takeSample(1)[0] in d
True
```

toLocalIterator()
Returns an iterator over the dataset.

Example:

```
>>> from pysparkling import Context
>>> sum(Context().parallelize([4, 9, 7, 3, 2, 5], 3).toLocalIterator())
30
```

top(*num*, *key=None*)
Top N elements in descending order.

Parameters

- **num** (*int*) – number of elements
- **key** – optional key function

Return type list

Example:

```
>>> from pysparkling import Context
>>> r = Context().parallelize([4, 9, 7, 3, 2, 5], 3)
>>> r.top(2)
[9, 7]
```

union(*other*)

Parameters **other** (RDD) – The other RDD for the union.

Return type *RDD*

Example:

```
>>> from pysparkling import Context
>>> my_rdd = Context().parallelize([4, 9, 7, 3, 2, 5], 3)
>>> my_rdd.union(my_rdd).count()
12
```

values()
Values of a (key, value) dataset.

Return type *RDD*

variance()
The variance of the dataset.

Return type float

Example:

```
>>> from pysparkling import Context
>>> Context().parallelize([1.5, 2.5]).variance()
0.25
```

zip(*other*)

Parameters **other** (RDD) – Other dataset to zip with.

Return type *RDD*

Note: Creating the new RDD is currently implemented as a local operation.

Example:

```
>>> from pysparkling import Context
>>> my_rdd = Context().parallelize([4, 9, 7, 3, 2, 5], 3)
>>> my_rdd.zip(my_rdd).collect()
[(4, 4), (9, 9), (7, 7), (3, 3), (2, 2), (5, 5)]
```

zipWithIndex()

Returns pairs of an original element and its index.

Return type *RDD*

Note: Creating the new RDD is currently implemented as a local operation.

Example:

```
>>> from pysparkling import Context
>>> my_rdd = Context().parallelize([4, 9, 7, 3, 2, 5], 3)
>>> my_rdd.zipWithIndex().collect()
[(4, 0), (9, 1), (7, 2), (3, 3), (2, 4), (5, 5)]
```

zipWithUniqueId()

Zip every entry with a unique index.

This is a fast operation.

Return type *RDD*

Example:

```
>>> from pysparkling import Context
>>> my_rdd = Context().parallelize([423, 234, 986, 5, 345], 3)
>>> my_rdd.zipWithUniqueId().collect()
[(423, 0), (234, 1), (986, 4), (5, 2), (345, 5)]
```

class pysparkling.**StatCounter**(*values=None*)

4.1.2 Context

A `Context` describes the setup. Instantiating a Context with the default arguments using `Context()` is the most lightweight setup. All data is just in the local thread and is never serialized or deserialized.

If you want to process the data in parallel, you can use the `multiprocessing` module. Given the limitations of the default `pickle` serializer, you can specify to serialize all methods with `cloudpickle` instead. For example, a common instantiation with `multiprocessing` looks like this:

```
c = Context(
    multiprocessing.Pool(4),
    serializer=cloudpickle.dumps,
    deserializer=pickle.loads,
)
```

This assumes that your data is serializable with `pickle` which is generally faster. You can also specify a custom serializer/deserializer for data.

class `pysparkling.`**`Context`**(*pool=None*, *serializer=None*, *deserializer=None*, *data_serializer=None*, *data_deserializer=None*)

Context object similar to a Spark Context.

The variable *_stats* contains measured timing information about data and function (de)serialization and workload execution to benchmark your jobs.

Parameters

- **`pool`** – An instance with a `map(func, iterable)` method.
- **`serializer`** – Serializer for functions. Examples are `pickle.dumps` and `dill.dumps`.
- **`deserializer`** – Deserializer for functions. Examples are `pickle.loads` and `dill.loads`.
- **`data_serializer`** – Serializer for the data.
- **`data_deserializer`** – Deserializer for the data.

`parallelize`(*x*, *numPartitions=None*)

parallelize x

Parameters

- **`x`** – An iterable (e.g. a list) that represents the data.
- **`numPartitions`** – (optional) The number of partitions the data should be split into. A partition is a unit of data that is processed at a time. Can be `None`.

Returns New RDD.

`pickleFile`(*name*, *minPartitions=None*)

read a pickle file

Reads files created with `RDD.saveAsPickleFile()` into an RDD.

Parameters

- **`name`** – Location of a file. Can include schemes like `http://`, `s3://` and `file://`, wildcard characters `?` and `*` and multiple expressions separated by `,`.
- **`minPartitions`** – (optional) By default, every file is a partition, but this option allows to split these further.

Returns New RDD.

Example with a serialized list:

```
>>> import pickle
>>> from pysparkling import Context
>>> from tempfile import NamedTemporaryFile
>>> tmpFile = NamedTemporaryFile(delete=True)
>>> tmpFile.close()
>>> with open(tmpFile.name, 'wb') as f:
```

```
...     pickle.dump(['hello', 'world'], f)
>>> Context().pickleFile(tmpFile.name).collect()[0] == 'hello'
True
```

runJob(*rdd*, *func*, *partitions=None*, *allowLocal=False*, *resultHandler=None*)
This function is used by methods in the RDD.

Note that the maps are only inside generators and the resultHandler needs to take care of executing the ones that it needs. In other words, if you need everything to be executed, the resultHandler needs to be at least `lambda x: list(x)` to trigger execution of the generators.

Parameters

- **func** – Map function. The signature is func(TaskContext, Iterator over elements).
- **partitions** – (optional) List of partitions that are involved. Default is `None`, meaning the map job is applied to all partitions.
- **allowLocal** – (optional) Allows for local execution. Default is False.
- **resultHandler** – (optional) Process the result from the maps.

Returns Result of resultHandler.

textFile(*filename*, *minPartitions=None*, *use_unicode=True*)
Read a text file into an RDD.

Parameters

- **filename** – Location of a file. Can include schemes like `http://`, `s3://` and `file://`, wildcard characters `?` and `*` and multiple expressions separated by `,`.
- **minPartitions** – (optional) By default, every file is a partition, but this option allows to split these further.
- **use_unicode** – (optional, default=True) Use `utf8` if `True` and `ascii` if `False`.

Returns New RDD.

union(*rdds*)

Parameters **rdds** – Iterable of RDDs.

Returns New RDD.

wholeTextFiles(*path*, *minPartitions=None*, *use_unicode=True*)
Read text files into an RDD of pairs of file name and file content.

Parameters

- **path** – Location of the files. Can include schemes like `http://`, `s3://` and `file://`, wildcard characters `?` and `*` and multiple expressions separated by `,`.
- **minPartitions** – (optional) By default, every file is a partition, but this option allows to split these further.
- **use_unicode** – (optional, default=True) Use `utf8` if `True` and `ascii` if `False`.

Returns New RDD.

4.1.3 fileio

The functionality provided by this module is used in `Context.textFile()` for reading and in `RDD.saveAsTextFile()` for writing. You can use this submodule for writing files directly with

`File(filename).dump(some_data)`, `File(filename).load()` and `File.exists(path)` to read, write and check for existance of a file. All methods transparently handle `http://`, `s3://` and `file://` locations and compression/decompression of `.gz` and `.bz2` files.

Use environment variables `AWS_SECRET_ACCESS_KEY` and `AWS_ACCESS_KEY_ID` for auth and use file paths of the form `s3://bucket_name/filename.txt`.

class `pysparkling.fileio.`**`File`**(*file_name*)
file object

Parameters **`file_name`** – Any file name. Supports the schemes `http://`, `s3://` and `file://`.

`dump`(*stream=None*)
Writes a stream to a file.

Parameters **`stream`** – A BytesIO instance. `bytes` are also possible and are converted to BytesIO.

Returns self

`exists`()
Checks both for a file or directory at this location.

Returns True or false.

`load`()
Load the data from a file.

Returns A `io.BytesIO` instance. Use `getvalue()` to get a string.

`make_public`(*recursive=False*)
Makes the file public. Currently only supported on S3.

Parameters **`recursive`** – Whether to apply this recursively.

static **`resolve_filenames`**(*all_expr*)
resolve expression for a filename

Parameters **`all_expr`** – A comma separated list of expressions. The expressions can contain the wildcard characters `*` and `?`. It also resolves Spark datasets to the paths of the individual partitions (i.e. `my_data` gets resolved to `[my_data/part-00000, my_data/part-00001]`).

Returns A list of file names.

class `pysparkling.fileio.`**`TextFile`**(*file_name*)
Derived from *pysparkling.fileio.File*.

Parameters **`file_name`** – Any text file name. Supports the schemes `http://`, `s3://` and `file://`.

`dump`(*stream=None*, *encoding=u'utf8'*, *encoding_errors=u'ignore'*)
Writes a stream to a file.

Parameters

- **`stream`** – An `io.StringIO` instance. A `basestring` is also possible and get converted to `io.StringIO`.
- **`encoding`** – (optional) The character encoding of the file.

Returns self

load(*encoding=u'utf8'*, *encoding_errors=u'ignore'*)
Load the data from a file.

Parameters **encoding** – (optional) The character encoding of the file.

Returns An `io.StringIO` instance. Use `read()` to get a string.

4.2 Parallelization

Pysparkling supports parallelizations on the local machine and across clusters of computers.

4.2.1 Threads and Processes

Single machine parallelization either with `concurrent.futures.ThreadPoolExecutor`, `concurrent.futures.ProcessPoolExecutor` and `multiprocessing.Pool` is supported.

4.2.2 ipcluster and IPython.parallel

Local test setup:

```
ipcluster start --n=2
```

```
from IPython.parallel import Client

c = Client()
print(c[:].map(lambda _: 'hello world', range(2)).get())
```

which should print `['hello world', 'hello world']`.

To run on a cluster, create a profile:

```
ipython profile create --parallel --profile=smallcluster

# start controller:
# Creates ~/.ipython/profile_smallcluster/security/ipcontroller-engine.json
# which is used by the engines to identify the location of this controller.
# This is the local-only IP address. Substitute with the machines IP
# address so that the engines can find it.
ipcontroller --ip=127.0.0.1 --port=7123 --profile=smallcluster

# start engines (assuming they have access to the
# ipcontroller-engine.json file)
ipengine --profile=smallcluster
```

Test it in Python:

```
from IPython.parallel import Client

c = Client(profile='smallcluster')
print(c[:].map(lambda _: 'hello world', range(2)).get())
```

If you don't want to start the engines manually, `ipcluster` comes with "Launchers" that can start them for you: https://ipython.org/ipython-doc/dev/parallel/parallel_process.html#using-ipcluster-in-ssh-mode

4.2.3 StarCluster

Setting up StarCluster was an experiment. However it does not integrate well with the rest of our EC2 infrastructure, so we switched to a Chef based setup where we use `ipcluster` directly. A blocker was that the number of engines per node is not configurable and we have many map jobs that wait on external responses.

Setup

```
# install
pip install starcluster

# create configuration
starcluster help  # choose the option to create a sample config file

# add your user id, aws_access_key_id and aws_secret_access_key to config

# create an ssh key (this creates a new key just for starcluster)
# and registers it with AWS
starcluster createkey starclusterkey -o ~/.ssh/starclusterkey.rsa

# add this key to config:
[key starclusterkey]
KEY_LOCATION=~/.ssh/starclusterkey.rsa
# and use this key in the cluster setup:
KEYNAME = starclusterkey

# disable the queue, Sun Grid Engine
# (unnecessary for pysparkling and takes time during setup)
DISABLE_QUEUE=True

# to enable IPython parallel support, uncomment these lines in config:
[plugin ipcluster]
SETUP_CLASS = starcluster.plugins.ipcluster.IPCluster

# and make sure you have this line inside the cluster section
[cluster smallcluster]
PLUGINS = ipcluster

# start the cluster
starcluster start smallcluster

# check it has started
starcluster listclusters
```

Currently use: `ami-da180db2` (Ubuntu 14.04 with 100GB EBS) on `m3.medium` instances.

Workarounds:

```
# this seems to be a dependency that does not get installed
pip install pexpect

# to validate the ssh host, you need to log in once manually, to add it
# to the list of known hosts
starcluster sshmaster smallcluster
```

In Python, you should now be able to run

```
from IPython.parallel import Client

# the exact command is printed after the cluster started
```

```
rc = Client('/Users/sven/.starcluster/ipcluster/SecurityGroup:@sc-smallcluster-us-east-1.json',
            sshkey='/Users/sven/.ssh/starclusterkey.rsa', packer='pickle')

view = rc[:]
results = view.map(lambda x: x**30, range(8))
print results.get()
```

which is also in `tests/starcluster_simple.py`.

Install your own software that is not on pypi:

```
pip install wheel
python setup.py bdist_wheel  # add --universal for Python2 and 3 packages
starcluster put smallcluster dist/your_package_name.whl /home/sgeadmin/your_package_name.whl

# ssh into remote machine
starcluster sshmaster smallcluster
> pip install --upgrade pip
> pip install wheel
> pip2.7 install /home/sgeadmin/your_package_name.whl
```

4.3 Development

Fork the Github repository and apply your changes in a feature branch. To run pysparkling's unit tests:

```
# install
pip install -e .[hdfs,tests]
flake8 --install-hook

# run linting and test
flake8
nosetests -vv
```

Don't run `python setup.py test` as this will not execute the doctests. When all tests pass, create a Pull Request on GitHub. Please also update `HISTORY.rst` with short description of your change.

To preview the docs locally, install the extra dependencies with `pip install -r docs/requirements.txt`, and then cd into `docs/sphinx`, run `make html` and open `docs/sphincs/_build/html/index.html`.

Please also try not to add derivative work from other projects. If you do, incorporate proper handling of external licenses in your Pull Request.