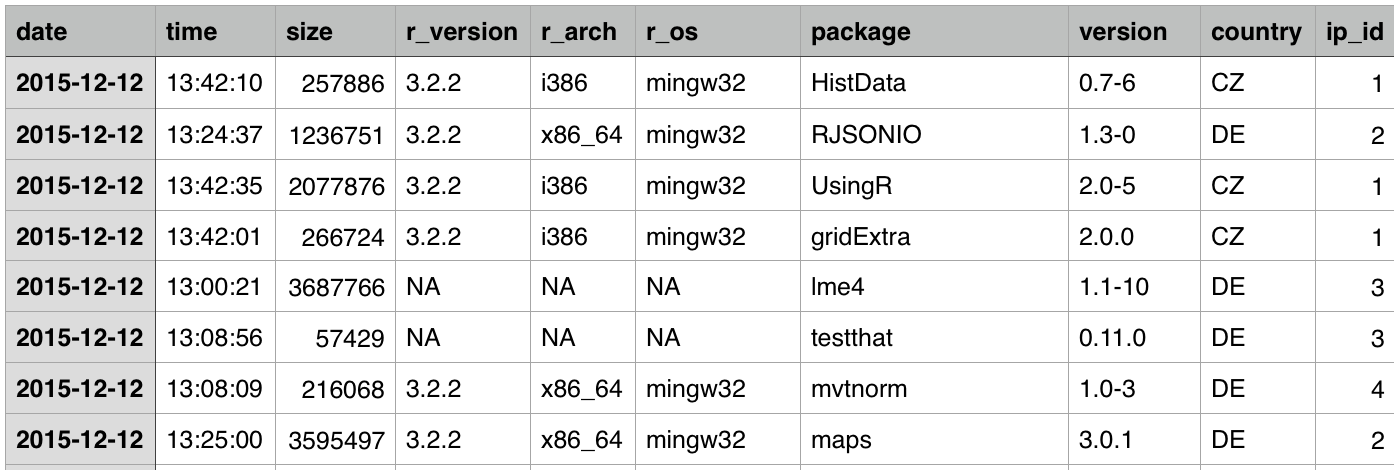
**Spark Practice**

In this exercise we will use Spark (PySpark) to look into a downloading log file in .CSV format and run some basic analysis on the data.

The material in this document is adapted from <https://github.com/XD-DENG/Spark-practice> For a more detailed information, please refer to the source.

**Sample Data**

The sample data we use here is from http://cran-logs.rstudio.com/. It is the full downloads log of R packages from Rstudio's CRAN mirror on December 12 2015. You can get the data for different dates from the link above.



We will use Spark to do some simple analytics on this data.

**Load Data**

The most common method used to load data is textFile. This method takes an URI for the file (local file or other URI like hdfs://), and will read the data in as a collections of lines.

# Load the data

>>> raw\_content = sc.textFile("2015-12-12.csv")

# Print the type of the object

>>> type(raw\_content)

<class 'pyspark.rdd.RDD'>

# Print the number of lines

>>> raw\_content.count()

421970

You may want to take note of that all of Spark’s file-based input methods, including textFile, support running on directories, compressed files, and wildcards as well. For example, you can use textFile("/my/directory"), textFile("/my/directory/.txt"), and textFile("/my/directory/.gz"). In our case, the two commands below will help load exactly the same data.

>>> a = sc.textFile("2015-12-12.csv")

>>> b = sc.textFile("2015-12-12.csv.gz")

>>> a.count()

421970

>>> b.count()

421970

This feature also makes things much simpler when we have multiple text data files to load. By giving the directory under where these files are ("/my/directory"), we can load many data files with only one line. Additionally, we can also specify the file types we would like to load, like with textFile("/my/directory/\*.txt"), we will only load those files with .txt file type in the directory we specified.

**Show the Head (First n rows)**

We can use take method to return first n rows.

>>> raw\_content.take(5)

[u'"date","time","size","r\_version","r\_arch","r\_os","package","version","country","ip\_id"',

u'"2015-12-12","13:42:10",257886,"3.2.2","i386","mingw32","HistData","0.7-6","CZ",1',

u'"2015-12-12","13:24:37",1236751,"3.2.2","x86\_64","mingw32","RJSONIO","1.3-0","DE",2',

u'"2015-12-12","13:42:35",2077876,"3.2.2","i386","mingw32","UsingR","2.0-5","CZ",1',

u'"2015-12-12","13:42:01",266724,"3.2.2","i386","mingw32","gridExtra","2.0.0","CZ",1']

We can also take samples randomly with takeSample method. With takeSample method, there are three arguments and we need to give at least two of them. They are "if replacement", "number of samples", and "seed" (optional).

>>> raw\_content.takeSample(True, 5, 3)

[u'"2015-12-12","16:41:22",18773,"3.2.3","x86\_64","mingw32","evaluate","0.8","US",10935',

u'"2015-12-12","13:06:32",494138,"3.2.3","x86\_64","linux-gnu","rjson","0.2.15","KR",655',

u'"2015-12-12","03:50:05",140207,NA,NA,NA,"SACOBRA","0.7","DE",129',

u'"2015-12-12","21:40:13",622505,"3.2.3","x86\_64","linux-gnu","stratification","2.2-5","US",4860',

u'"2015-12-12","23:52:06",805204,"3.2.2","x86\_64","mingw32","readxl","0.1.0","CA",104']

If we specify the last argument, i.e. seed, then we can reproduce the samples exactly.

**Transformation (map & flatMap)**

We may note that each row of the data is a character string, and it would be more convenient to have an array instead. So we use map to transform them and use take method to get the first three rows to check how the results look like.

>>> content = raw\_content.map(lambda x: x.split(','))

>>> content.take(3)

[

[u'"date"', u'"time"', u'"size"', u'"r\_version"', u'"r\_arch"', u'"r\_os"', u'"package"', u'"version"', u'"country"', u'"ip\_id"'],

[u'"2015-12-12"', u'"13:42:10"', u'257886', u'"3.2.2"', u'"i386"', u'"mingw32"', u'"HistData"', u'"0.7-6"', u'"CZ"', u'1'],

[u'"2015-12-12"', u'"13:24:37"', u'1236751', u'"3.2.2"', u'"x86\_64"', u'"mingw32"', u'"RJSONIO"', u'"1.3-0"', u'"DE"', u'2']

]

I would say map(function) method is one of the most basic and important methods in Spark. It returns a new distributed dataset formed by passing each element of the source through a function specified by user.

You may have noted that there is another method named flatMap. Then what's the difference between map and flatMap? We can look into a simple example first.

>>> text=["a b c", "d e", "f g h"]

>>> sc.parallelize(text).map(lambda x:x.split(" ")).collect()

[['a', 'b', 'c'], ['d', 'e'], ['f', 'g', 'h']]

>>> sc.parallelize(text).flatMap(lambda x:x.split(" ")).collect()

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

To put it simply (maybe not precise), we can say that map will return a sequence of the same length as the original data. In this sequence each element is a sub-sequence corresponding to one element in original data. flatMap will return a sequence whose length equals to the sum of the lengths of all sub-sequance returned by map.

**Reduce and Counting**

Here we would like to know how many download records each package has. For example, for R package "Rcpp", we want to know how many rows belong to it.

# Note here x[6] is just the 7th element of each row, that is the package name.

>>> package\_count = content.map(lambda x: (x[6], 1)).reduceByKey(lambda a,b: a+b)

>>> type(package\_count)

<class 'pyspark.rdd.PipelinedRDD'>

>>> package\_count.count()

8660

>>> package\_count.take(5)

[(u'SIS', 24),

(u'StatMethRank', 15),

(u'dbmss', 54),

(u'searchable', 14),

(u'RcmdrPlugin.TextMining', 3)]

To achive the same purpose, we can also use countByKey method. The result returned by it is in hashmap (like dictionary) structure.

>>> package\_count\_2 = content.map(lambda x: (x[6], 1)).countByKey()

>>> type(package\_count\_2)

<type 'collections.defaultdict'>

>>> package\_count\_2['ggplot2']

3913

>>> package\_count\_2['stm']

25

Please note that countByKey method ONLY works on RDDs of type (K, V), returning a hashmap of (K, int) pairs with the COUNT of each key [1]. And the value of V will NOT affect the result! Just like the example below.

>>> package\_count\_2 = content.map(lambda x: (x[6], 1)).countByKey()

>>> package\_count\_2['ggplot2']

3913

>>> package\_count\_2\_1 = content.map(lambda x: (x[6], 3)).countByKey()

>>> package\_count\_2\_1['ggplot2']

3913

>>> package\_count\_2\_2 = content.map(lambda x: (x[6], "test")).countByKey()

>>> package\_count\_2\_2['ggplot2']

3913

**Sorting**

After counting by reduce method, we may want to know the rankings of these packages based on how many downloads they have. Then we need to use sortByKey method. Please note:

The 'Key' here refers to the first element of each array.

The argument of sortByKey (0 or 1) will determine if we're sorting descending ('0') or ascending ('1').

# Sort DESCENDING and get the first 10

>>> package\_count.map(lambda x: (x[1], x[0])).sortByKey(0).take(10)

[(4783, u'Rcpp'),

(3913, u'ggplot2'),

(3748, u'stringi'),

(3449, u'stringr'),

(3436, u'plyr'),

(3265, u'magrittr'),

(3223, u'digest'),

(3205, u'reshape2'),

(3046, u'RColorBrewer'),

(3007, u'scales')]

# Sort ASCENDING and get the first 10

>>> package\_count.map(lambda x: (x[1], x[0])).sortByKey(1).take(10)

[(1, u'TSjson'),

(1, u'ebayesthresh'),

(1, u'parspatstat'),

(1, u'gppois'),

(1, u'JMLSD'),

(1, u'kBestShortestPaths'),

(1, u'StVAR'),

(1, u'mosaicManip'),

(1, u'em2'),

(1, u'DART')]

Other than sorting by key (normally it's the first element in each observation), we can also specify by which element to sort using method sortBy,

# For the sortBy method, default value for ascending is True

>>> package\_count.sortBy(lambda x:x[1]).take(5)

[(u'TSjson', 1),

(u'ebayesthresh', 1),

(u'parspatstat', 1),

(u'gppois', 1),

(u'JMLSD', 1)]

>>> package\_count.sortBy(lambda x:x[1], ascending = False).take(5)

[(u'Rcpp', 4783),

(u'ggplot2', 3913),

(u'stringi', 3748),

(u'stringr', 3449),

(u'plyr', 3436)]

**Filter**

We can consider filter as the SELECT \* from TABLE WHERE ??? statement in SQL. It can help return a new dataset formed by selecting those elements of the source on which the function specified by user returns true.

For example, we would want to obtain these downloading records of R package "Rtts" from China (CN), then the condition is "package == 'Rtts' AND country == 'CN'".

>>> content.filter(lambda x: x[6] == 'Rtts' and x[8] == 'CN').count()

1

>>> content.filter(lambda x: x[6] == 'Rtts' and x[8] == 'CN').take(1)

[[u'2015-12-12', u'20:15:24', u'23820', u'3.2.2', u'x86\_64', u'mingw32', u'Rtts', u'0.3.3', u'CN', u'41']]

**Collect Result ('Export' into Python)**

All the operations I listed above were done as RDD (Resilient Distributed Datasets). We can say that they were implemented 'within' Spark. And we may want to transfer some dataset into Python itself.

take method we used above can help us fulfill this purpose partially. But we also have collect method to do this, and the difference between collect and take is that the former will return all the elements in the dataset by default and the later one will return the first n rows (n is specified by user).

>>> temp = content.filter(lambda x: x[6] == 'Rtts' and x[8] == 'US').collect()

>>> type(temp)

<type 'list'>

>>> temp

[

[u'2015-12-12', u'04:52:36', u'23820', u'3.2.3', u'i386', u'mingw32', u'Rtts', u'0.3.3', u'US', u'1652'],

[u'2015-12-12', u'20:31:45', u'23820', u'3.2.3', u'x86\_64', u'linux-gnu', u'Rtts', u'0.3.3', u'US', u'4438']

]

**Set Operation**

Like the set operators in SQL, we can do set operations in Spark. Here we would introduce union, intersection, and distinct. We can make intuitive interpretations as below.

union of A and B: return elements of A AND elements of B.

intersection of A and B: return these elements existing in both A and B.

distinct of A: return the distinct values in A. That is, if element a appears more than once, it will only appear once in the result returned.

>>> raw\_content.count()

421970

# one set's union with itself equals to its "double"

>>> raw\_content.union(raw\_content).count()

843940

# one set's intersection with itself equals to its disctinct value set

>>> raw\_content.intersection(raw\_content).count()

421553

>>> raw\_content.distinct().count()

421553

One point we need to take note of is that if each line of our data is an array instead of a string, intersection and distinct methods can't work properly. This is why we used raw\_content instead of content here as example.

**Join**

The data process methods in Spark are quite similar to that in SQL, like we can use join method in Spark, which is great news! Outer joins are also supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin. Additionally, Cartesian product is available as well (please note Spark SQL is available for similar purpose and would be preferred & recommended).

When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key.

# generate a new RDD in which the 'country' variable is KEY

>>> content\_modified=content.map(lambda x:(x[8], x))

# create a mapping table of the abbreviations of four countries and their full names.

>>> mapping=[('DE', 'Germany'), ('US', 'United States'), ('CN', 'China'), ('IN',"India")]

>>> mapping=sc.parallelize(mapping)

# join

>>> content\_modified.join(mapping).takeSample(False, 8)

[

(u'CN', ([u'2015-12-12', u'19:26:01', u'512', u'NA', u'NA', u'NA', u'reweight', u'1.01', u'CN', u'4721'], 'China')),

(u'US', ([u'2015-12-12', u'18:15:11', u'14271399', u'3.2.1', u'x86\_64', u'mingw32', u'stringi', u'1.0-1', u'US', u'11837'], 'United States')),

(u'US', ([u'2015-12-12', u'00:03:27', u'392370', u'3.2.3', u'x86\_64', u'linux-gnu', u'colorspace', u'1.2-6', u'US', u'12607'], 'United States')),

(u'US', ([u'2015-12-12', u'05:10:29', u'290932', u'3.2.2', u'x86\_64', u'mingw32', u'iterators', u'1.0.8', u'US', u'5656'], 'United States')),

(u'US', ([u'2015-12-12', u'22:28:47', u'2143454', u'3.2.3', u'x86\_64', u'linux-gnu', u'quantreg', u'5.19', u'US', u'16318'], 'United States')),

(u'US', ([u'2015-12-12', u'13:12:26', u'985806', u'3.2.3', u'x86\_64', u'linux-gnu', u'plotly', u'2.0.3', u'US', u'2570'], 'United States')),

(u'CN', ([u'2015-12-12', u'17:04:44', u'178399', u'3.2.1', u'x86\_64', u'darwin13.4.0', u'apsrtable', u'0.8-8', u'CN', u'41'], 'China')),

(u'US', ([u'2015-12-12', u'06:41:09', u'76007', u'3.2.3', u'i386', u'mingw32', u'superpc', u'1.09', u'US', u'1985'], 'United States'))

]

# left outer join.

In the mapping table, we provided the mappings for only four countries, so we find some 'None' values in the returned result below.

>>> content\_modified.leftOuterJoin(mapping).takeSample(False, 8)

[

(u'US', ([u'2015-12-12', u'15:43:03', u'153892', u'3.2.2', u'i386', u'mingw32', u'gridBase', u'0.4-7', u'US', u'8922'], 'United States')),

(u'CN', ([u'2015-12-12', u'19:59:37', u'82833', u'3.2.3', u'x86\_64', u'mingw32', u'rgcvpack', u'0.1-4', u'CN', u'41'], 'China')),

(u'JP', ([u'2015-12-12', u'17:24:59', u'2677787', u'3.2.3', u'i386', u'mingw32', u'ggplot2', u'1.0.1', u'JP', u'3597'], None)),

(u'TN', ([u'2015-12-12', u'13:40:13', u'1229084', u'3.2.2', u'x86\_64', u'mingw32', u'forecast', u'6.2', u'TN', u'10847'], None)),

(u'US', ([u'2015-12-12', u'05:09:59', u'75327', u'3.2.3', u'x86\_64', u'mingw32', u'xml2', u'0.1.2', u'US', u'5530'], 'United States')),

(u'AE', ([u'2015-12-12', u'14:23:56', u'695625', u'3.1.2', u'i386', u'mingw32', u'mbbefd', u'0.7', u'AE', u'556'], None)),

(u'KR', ([u'2015-12-12', u'16:31:34', u'36701', u'3.2.3', u'x86\_64', u'linux-gnu', u'ttScreening', u'1.5', u'KR', u'4986'], None)),

(u'US', ([u'2015-12-12', u'15:43:08', u'35212', u'3.2.2', u'x86\_64', u'mingw32', u'reshape2', u'1.4.1', u'US', u'8922'], 'United States'))]

**Caching**

Some RDDs may be repeatedly accessed, like the RDD content in the example above. In such situation, we may want to pull such RDDs into cluster-wide in-memory cache so that the computing relating to them will not be repeatedly implemented, which can help save resource and time. This is called "caching" in Spark, and can be done using RDD.cache() or RDD.persist() method.

Spark automatically monitors cache usage on each node and drops out old data partitions in a least-recently-used (LRU) fashion. Of course we can also manually remove an RDD instead of waiting for it to fall out of the cache, using the RDD.unpersist() method.

We can also use .is\_cached to check whether a RDD is already cached or not.

>>> content.cache()

>>> content.is\_cached

True

>>> content.unpersist()

>>> content.is\_cached

False

Please note caching may make little or even no difference when the data is small. But it will be significantly efficient when we're trying to handle big-size data in distributed fashion.