PyRDF

Release 0.1

Shravan Murali; Enric Tejedor Saavedra; Enrico Guiraud; Diogo C

Apr 07, 2020

CONTENTS:

A pythonic wrapper around ROOT's RDataFrame with support for distributed execution.

Sample usage:

```
import PyRDF, ROOT
PyRDF.use('spark', {'npartitions':4})

df = PyRDF.RDataFrame("data", ['https://root.cern/files/teaching/CMS_Open_Dataset.root
↪',])

etaCutStr = "fabs(eta1) < 2.3"
df_f = df.Filter(etaCutStr)

df_histogram = df_f.Histo1D("eta1")

canvas = ROOT.TCanvas()
df_histogram.Draw()
canvas.Draw()
```

CHAPTER

ONE

THE PYRDF API REFERENCE

PyRDF.**create_logger**(*level='WARNING'*, *log_path='./PyRDF.log'*)
PyRDF basic logger

PyRDF.**include_headers**(*headers_paths*)
Includes the C++ headers to be declared before execution. Each header is also declared on the current running session.

Parameters **headers_paths** (*str, iter*) – A string or an iterable (such as a list, set...) containing the paths to all necessary C++ headers as strings. This function accepts both paths to the headers themselves and paths to directories containing the headers.

PyRDF.**include_shared_libraries**(*shared_libraries_paths*)
Includes the C++ shared libraries to be declared before execution. Each library is also declared on the current running session. If any pcm file is present in the same folder as the shared libraries, the function will try to retrieve them (and distribute them if working on a distributed backend).

Parameters **shared_libraries_paths** (*str, iter*) – A string or an iterable (such as a list, set...) containing the paths to all necessary C++ shared libraries as strings. This function accepts both paths to the libraries themselves and paths to directories containing the libraries.

PyRDF.**initialize**(*fun*, **args*, ***kwargs*)
Set a function that will be executed as a first step on every backend before any other operation. This method also executes the function on the current user environment so changes are visible on the running session.

This allows users to inject and execute custom code on the worker environment without being part of the RDataFrame computational graph.

Parameters

- **fun** (*function*) – Function to be executed.
- ***args** (*list*) – Variable length argument list used to execute the function.
- ****kwargs** (*dict*) – Keyword arguments used to execute the function.

PyRDF.**send_generic_files**(*files_paths*)
Sends to the workers the generic files needed by the user.

Parameters **files_paths** (*str, iter*) – Paths to the files to be sent to the distributed workers.

PyRDF.**use**(*backend_name*, *conf={}*)
Allows the user to choose the execution backend.

Parameters

- **backend_name** (*str*) – This is the name of the chosen backend.
- **conf** (*str, optional*) – This should be a dictionary with necessary configuration parameters. Its default value is an empty dictionary {}.

1.1 The CallableGenerator module

class PyRDF.CallableGenerator.**CallableGenerator**(*head_node*)
Class that generates a callable to parse a PyRDF graph.

head_node
Head node of a PyRDF graph.

__init__(*head_node*)
Creates a new *CallableGenerator*.

Parameters **head_node** – Head node of a PyRDF graph.

get_action_nodes(*node_py=None*)
Recurses through PyRDF graph and collects the PyRDF node objects.

Parameters **node_py** (*optional*) – The current state's PyRDF node. If *None*, it takes the value of *self.head_node*.

Returns A list of the action nodes of the graph in DFS order, which coincides with the order of execution in the callable function.

Return type list

get_callable()
Converts a given graph into a callable and returns the same.

Returns The callable that takes in a PyROOT RDataFrame object and executes all operations from the PyRDF graph on it, recursively.

Return type function

1.2 The Node module

class PyRDF.Node.**Node**(*get_head*, *operation*, **args*)
A Class that represents a node in RDataFrame operations graph. A Node houses an operation and has references to children nodes. For details on the types of operations supported, try :

Example:

```
import PyRDF
PyRDF.use(...) # Choose your backend
print(PyRDF.current_backend.supported_operations)
```

get_head
A lambda function that returns the head node of the current graph.

Type function

operation
The operation that this Node represents. This could be None.

children
A list of *PyRDF.Node* objects which represent the children nodes connected to the current node.

Type list

_new_op_name
The name of the new incoming operation of the next child, which is the last child node among the current node's children.

Type str

value
The computed value after executing the operation in the current node for a particular PyRDF graph. This is permanently `None` for transformation nodes and the action nodes get a `ROOT.RResultPtr` after event-loop execution.

pyroot_node
Reference to the PyROOT object that implements the functionality of this node on the cpp side.

has_user_references
A flag to check whether the node has direct user references, that is if it is assigned to a variable. Default value is `True`, turns to `False` if the proxy that wraps the node gets garbage collected by Python.

Type bool

__getstate__()
Converts the state of the current node to a Python dictionary.

Returns A dictionary that stores all instance variables that represent the current PyRDF node.

Return type dictionary

__init__(*get_head*, *operation*, **args*)
Creates a new node based on the operation passed as argument.

Parameters

- **get_head** (*function*) – A lambda function that returns the head node of the current graph. This value could be *None*.
- **operation** (PyRDF.Operation.Operation) – The operation that this Node represents. This could be `None`.

__setstate__(*state*)
Retrieves the state dictionary of the current node and sets the instance variables.

Parameters **state** (*dict*) – This is the state dictionary that needs to be converted to a *Node* object.

graph_prune()
Prunes nodes from the current PyRDF graph under certain conditions. The current node will be pruned if it has no children and the user application does not hold any reference to it. The children of the current node will get recursively pruned.

Returns True if the current node has to be pruned, False otherwise.

Return type bool

is_prunable()
Checks whether the current node can be pruned from the computational graph.

Returns True if the node has no children and no user references or its value has already been computed, False otherwise.

Return type bool

1.3 The Operation module

class PyRDF.Operation.**Operation**(*name*, **args*, ***kwargs*)

A Generic representation of an operation. The operation could be a transformation or an action.

Types

A class member that is an `Enum` of the types of operations supported. This can be either `ACTION`, `TRANSFORMATION` or `INSTANT_ACTION`.

name

Name of the current operation.

Type str

args

Variable length argument list for the current operation.

Type list

kwargs

Arbitrary keyword arguments for the current operation.

Type dict

op_type

The type or category of the current operation (`ACTION`, `TRANSFORMATION` or `INSTANT_ACTION`).

For the list of operations that your current backend supports, try :

Example:

```
import PyRDF
PyRDF.use(...) # Choose a backend

print(PyRDF.current_backend.supported_operations)
```

class Types

An enumeration.

__init__(*name*, **args*, ***kwargs*)

Creates a new *Operation* for the given name and arguments.

Parameters **name** (*str*) – Name of the current operation.

args (list): Variable length argument list for the current operation.

kwargs (dict): Keyword arguments for the current operation.

is_action()

Checks if the current operation is an action.

Returns True if the current operation is an action, False otherwise.

Return type bool

is_instant_action()

Checks if the current operation is an instant action.

Returns

True if the current operation is an instant action, False otherwise.

Return type bool

is_transformation()
Checks if the current operation is a transformation.

Returns True if the current operation is a transformation, False otherwise.

Return type bool

1.4 The Proxy module

class PyRDF.Proxy.**ActionProxy**(*node*)
Instances of ActionProxy act as futures of the result produced by some action node. They implement a lazy synchronization mechanism, i.e., when they are accessed for the first time, they trigger the execution of the whole RDataFrame graph.

GetValue()
Returns the result value of the current action node if it was executed before, else triggers the execution of the entire PyRDF graph before returning the value.

Returns The value of the current action node, obtained after executing the current action node in the computational graph.

__getattr__(*attr*)
Intercepts calls on the result of the action node.

Returns A method to handle an operation call to the current action node.

Return type function

class PyRDF.Proxy.**Proxy**(*node*)
Abstract class for proxies objects. These objects help to keep track of nodes' variable assignment. That is, when a node is no longer assigned to a variable by the user, the role of the proxy is to show that. This is done via changing the value of the `has_user_references` of the proxied node from `True` to `False`.

__del__()
This function is called right before the current Proxy gets deleted by Python. Its purpose is to show that the wrapped node has no more user references, which is one of the conditions for the node to be pruned from the computational graph.

abstract __getattr__(*attr*)
Proxies have to declare the way they intercept calls to attributes and methods of the proxied node.

__init__(*node*)
Creates a new *Proxy* object for a given node.

Parameters **proxied_node** – The node that the current Proxy should wrap.

class PyRDF.Proxy.**TransformationProxy**(*node*)
A proxy object to an non-action node. It implements acces to attributes and methods of the proxied node. It is also in charge of the creation of a new operation node in the graph.

__getattr__(*attr*)
Intercepts calls to attributes and methods of the proxied node and returns the appropriate object(s).

Parameters **attr** (*str*) – The name of the attribute or method of the proxied node the user wants to access.

1.5 The RDataFrame module

class PyRDF.RDataFrame.**HeadNode**(**args*)

The Python equivalent of ROOT C++'s RDataFrame class.

args

A list of arguments that were provided to construct the RDataFrame object.

Type list

PyRDF's RDataFrame constructor accepts the same arguments as the ROOT's RDataFrame constructor (see RDataFrame)

In addition, PyRDF allows you to use Python lists in place of C++ vectors as arguments of the constructor, example:

```
PyRDF.RDataFrame("myTree", ["file1.root", "file2.root"])
```

Raises ***RDataFrameException*** – An exception raised when input arguments to the RDataFrame constructor are incorrect.

__init__(**args*)

Creates a new RDataFrame instance for the given arguments.

Parameters ***args** (*list*) – Variable length argument list to construct the RDataFrame object.

get_branches()

Gets list of default branches if passed by the user.

get_inputfiles()

Get list of input files.

This list can be extracted from a given TChain or from the list of arguments.

Returns Name of a single file, list of files (both may contain globbing characters), or None if there are no input files.

Return type (str, list, None)

get_num_entries()

Gets the number of entries in the given dataset.

Returns This is the computed number of entries in the input dataset.

Return type int

get_tree()

Get ROOT.TTree instance used as an argument to PyRDF.RDataFrame()

Returns instance of the tree used to instantiate the RDataFrame, or *None* if another object was used. ROOT.Tchain inherits from ROOT.TTree so that can be the return value as well.

Return type (ROOT.TTree, None)

get_treename()

Get name of the TTree.

Returns Name of the TTree, or `None` if there is no tree.

Return type (str, None)

class PyRDF.RDataFrame.**RDataFrame**

User interface to the object containing the Python equivalent of ROOT C++'s RDataFrame class. The purpose of this class is to kickstart the head node of the computational graph, together with a proxy wrapping it.

static __new__(*cls*, **args*)

Creates the head node of the graph with the arguments provided by the user, then returns a proxy to that node.

Parameters ***args** (*list*) – A list of arguments that were provided by the user to construct the RDataFrame object.

exception PyRDF.RDataFrame.**RDataFrameException**(*exception*, *msg*)

A special type of Exception that shows up for incorrect arguments to RDataFrame.

__init__(*exception*, *msg*)

Creates a new *RDataFrameException*.

Parameters

- **exception** – An exception of type `Exception` or any child class of `Exception`.
- **msg** (*str*) – Message to be printed while raising exception.

CHAPTER

TWO

PYRDF'S SUPPORTED BACKENDS

2.1 The parent backend class

class PyRDF.backend.Backend.**Backend**(*config={}*)

Base class for RDataFrame backends. Subclasses of this class need to implement the 'execute' method.

supported_operations

List of operations supported by the backend.

Type list

initialization

Store user's initialization method, if defined.

Type function

__init__(*config={}*)

Creates a new instance of the desired implementation of *Backend*.

Parameters **config** (*dict*) – The config object for the required backend. The default value is an empty Python dictionary: `{}`.

check_supported(*operation_name*)

Checks if a given operation is supported by the given backend.

Parameters **operation_name** (*str*) – Name of the operation to be checked.

Raises

- **Exception** – This happens when *operation_name* doesn't exist
- **the supported_operations instance attribute.** –

abstract execute(*generator*)

Subclasses must define how to run the RDataFrame graph on a given environment.

classmethod register_initialization(*fun*, **args*, ***kwargs*)

Convert the initialization function and its arguments into a callable without arguments. This callable is saved on the backend parent class. Therefore, changes on the runtime backend do not require users to set the initialization function again.

Parameters

- **fun** (*function*) – Function to be executed.
- ***args** (*list*) – Variable length argument list used to execute the function.
- ****kwargs** (*dict*) – Keyword arguments used to execute the function.

2.2 The local backend

class PyRDF.backend.Local.**Local**(*config={}*)
Backend that relies on the C++ implementation of RDataFrame to locally execute the current graph.

config
The config object for the Local backend.

Type dict

__init__(*config={}*)
Creates a new instance of the Local implementation of *Backend*.

Parameters **config** (*dict, optional*) – The config object for the required backend. The default value is an empty Python dictionary: `{}`.

execute(*generator*)
Executes locally the current RDataFrame graph.

Parameters **generator** (*PyRDF.CallableGenerator*) – An instance of `CallableGenerator` that is responsible for generating the callable function.

2.3 The distributed backend parent class

class PyRDF.backend.Dist.**Dist**(*config={}*)
Base class for implementing all distributed backends.

npartitions
The number of chunks to divide the dataset in, each chunk is then processed in parallel.

Type int

supported_operations
list of supported RDataFrame operations in a distributed environment.

Type list

friend_info
A class instance that holds information about any friend trees of the main ROOT.TTree

Type PyRDF.Dist.FriendInfo

abstract ProcessAndMerge(*mapper*, *reducer*)
Subclasses must define how to run map-reduce functions on a given backend.

__init__(*config={}*)
Creates an instance of Dist.

Parameters **config** (*dict, optional*) – The config options for the current distributed backend. Default value is an empty python dictionary: `{}`.

build_ranges()
Define two type of ranges based on the arguments passed to the RDataFrame head node.

abstract distribute_files(*includes_list*)
Subclasses must define how to send all files needed for the analysis (like headers and libraries) to the workers.

execute(*generator*)
Executes the current RDataFrame graph in the given distributed environment.

Parameters **generator** (*PyRDF.CallableGenerator*) – An instance of `CallableGenerator` that is responsible for generating the callable function.

get_clusters(*treename*, *filelist*)
Extract a list of cluster boundaries for the given tree and files

Parameters

- **treename** (*str*) – Name of the TTree split into one or more files.
- **filelist** (*list*) – List of one or more ROOT files.

Returns List of tuples defining the cluster boundaries. Each tuple contains four elements: first entry of a cluster, last entry of cluster, offset of the cluster and file where the cluster belongs to.

Return type list

class PyRDF.backend.Dist.**FriendInfo**(*friend_names=[]*, *friend_file_names=[]*)
A simple class to hold information about friend trees.

friend_names
A list with the names of the *ROOT.TTree* objects which are friends of the main *ROOT.TTree*.

Type list

friend_file_names
A list with the paths to the files corresponding to the trees in the *friend_names* attribute. Each element of *friend_names* can correspond to multiple file names.

Type list

__bool__()
Define the behaviour of FriendInfo instance when boolean evaluated. Both lists have to be non-empty in order to return True.

Returns True if both lists are non-empty, False otherwise.

Return type bool

__init__(*friend_names=[]*, *friend_file_names=[]*)
Create an instance of FriendInfo

Parameters

- **friend_names** (*list*) – A list containing the treenames of the friend trees.
- **friend_file_names** (*list*) – A list containing the file names corresponding to a given treename in friend_names. Each treename can correspond to multiple file names.

__nonzero__()
Python 2 dunder method for __bool__. Kept for compatibility.

class PyRDF.backend.Dist.**Range**(*start*, *end*, *filelist=None*, *friend_info=None*)
Base class to represent ranges.

A range represents a logical partition of the entries of a chain and is the basis for parallelization. First entry of the range (start) is inclusive while the second one is not (end).

__init__(*start*, *end*, *filelist=None*, *friend_info=None*)
Create an instance of a Range

Parameters

- **start** (*int*) – First entry of the range.

- **end** (*int*) – Last entry of the range, which is exclusive.
- **filelist** (*list, optional*) – Files where the range of entries belongs to.

__repr__()
Return a string representation of the range composition.

2.4 The Spark distributed backend

class PyRDF.backend.Spark.**Spark**(*config={}*)
Backend that executes the computational graph using using *Spark* framework for distributed execution.

ProcessAndMerge(*mapper*, *reducer*)
Performs map-reduce using Spark framework.

Parameters

- **mapper** (*function*) – A function that runs the computational graph and returns a list of values.
- **reducer** (*function*) – A function that merges two lists that were returned by the mapper.

Returns A list representing the values of action nodes returned after computation (Map-Reduce).

Return type list

__init__(*config={}*)
Creates an instance of the Spark backend class.

Parameters **config** (*dict, optional*) – The config options for Spark backend. The default value is an empty Python dictionary `{}`. `config` should be a dictionary of Spark configuration options and their values with :obj:'npartitions' as the only allowed extra parameter.

Example:

```
config = {
    'npartitions':20,
    'spark.master':'myMasterURL',
    'spark.executor.instances':10,
    'spark.app.name':'mySparkAppName'
}
```

Note: If a SparkContext is already set in the current environment, the Spark configuration parameters from :obj:'config' will be ignored and the already existing SparkContext would be used.

distribute_files(*includes_list*)
Spark supports sending files to the executors via the *SparkContext.addFile* method. This method receives in input the path to the file (relative to the path of the current python session). The file is initially added to the Spark driver and then sent to the workers when they are initialized.

Parameters **includes_list** (*list*) – A list consisting of all necessary C++ files as strings, created one of the *include* functions of the PyRDF API.

CHAPTER

THREE

PYRDF'S UTILITY FUNCTIONS

class PyRDF.backend.Utils.**Utils**

Class that houses general utility functions.

classmethod declare_headers(*headers_to_include*)

Declares all required headers using the ROOT's C++ Interpreter.

Parameters **headers_to_include** (*list*) – This list should consist of all necessary C++ headers as strings.

classmethod declare_shared_libraries(*libraries_to_include*)

Declares all required shared libraries using the ROOT's C++ Interpreter.

Parameters **libraries_to_include** (*list*) – This list should consist of all necessary C++ shared libraries as strings.

classmethod extend_include_path(*include_path*)

Extends the list of paths in which ROOT looks for headers and libraries. Every header directory is added to the internal include path of ROOT so the interpreter can find them. Even if the same path is added twice, ROOT keeps a collection of unique paths. Find more at **`TInterpreter<https://root.cern.ch/doc/master/classTInterpreter.html>`_**

Parameters **include_path** (*str*) – the path to the directory containing files needed for the analysis.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

V