pygraphx – Python Bindings for GraphX

New in version Apache Spark™ 1.6.

Source code:

SPARK_HOME/python/pyspark/graphx/vertex.py
SPARK_HOME/python/ pyspark /graphx/edge.py
SPARK_HOME/python/ pyspark /graphx/graph.py
SPARK_HOME/graphx/src/main/scala/org/apache.spark/api/python/PythonVertexRDD.scala
SPARK_HOME/graphx/src/main/scala/org/apache.spark/api/python/PythonEdgeRDD.scala
SPARK_HOME/graphx/src/main/scala/org/apache.spark/api/python/PythonGraphRDD.scala

Test code:

Scala:

SPARK_HOME/graphx/src/test/scala/org/apache.spark/api/python/PythonAPIGraphSuite.scala
SPARK_HOME/graphx/src/test/scala/org/apache.spark/api/python/PythonAPIVertexSuite.scala
SPARK_HOME/graphx/src/test/scala/org/apache.spark/api/python/PythonAPIEdgeSuite.scala

Python:

SPARK_HOME/python/pyspark/graphx/tests.py

PyGraphX is a submodule of PySpark. It defines the Python bindings for VertexRDD, EdgeRDD and Graph RDD in Spark GraphX. In GraphX, a graph is defined in terms of vertices, edges and their properties. Vertices are tuples of *vertex id's* and *vertex property objects*. Each vertex has a unique id. Edges on the other hand are tuples of *source vertex id*, *destination vertex id* and *edge property object*. Graph RDD objects contain one vertex RDD and one edge RDD object. The objective of PyGraphX is to expose in Python all the transformations and actions and vertex programming available in Scala implementation of VertexRDD, EdgeRDD and GraphRDD. This module implements VertexRDD, EdgeRDD and GraphRDD data types.

VertexRDD	Data type for vertices in a graph.	New in version 1.6
EdgeRDD	Data type for edges in a graph	New in version 1.6
GraphRDD	Data type for graphs. Also exposes the Pregel API and library of standard algorithms also available in GraphX	New in version 1.6

VertexRDD objects

A VertexRDD class is provided to support convenient and rapid functions on a distributed list of vertices of a graph. For example, a vertex class with two string values as property can be used as:

```
>>> rdd = sc.parallelize([(1, (“Alice”, “manager”)), (2, (“Bob”, “engineer”))])
>>> vertices = VertexRDD(rdd)
>>> vertices.count()
>>> 2
```

class VertexRDD(parent-RDD, spark-context, deserializer):

A VertexRDD is an unordered indexed RDD of vertex tuples. The vertex tuples are made up of vertex ids of data type long and vertex property objects which can be of any standard Python data types. Vertices are created from parent RDDs as following:

```
>>> rdd = sc.parallelize([(1, (“Alice”, “professor”)), (2, (“Bob”, “postdoc”)), (3, (“Charlie”, “student”)), (4, (“David”, “student”))])
>>> # Create from a parent RDD using its context and deserializer
>>> vertices = VertexRDD(rdd)
>>> # Create from a parent RDD using a separate context
>>> vertices = VertexRDD(rdd, sc)
>>> # Create from a parent RDD using separate context and deserializer
>>> vertices = VertexRDD(rdd, sc, BatchSerializer(CloudPickleSerializer()))
```

In addition to the operations available in PySpark RDD objects, VertexRDD provides the following additional methods:

mapValues(f)

Map only the vertex properties of a VertexRDD. Keys remain unchanged. The argument “f” is a lambda function to be executed for every vertex in the RDD.

```
>>> rdd = sc.parallelize([(1, "b"), (2, "a"), (3, "c")])
>>> # Create from a parent RDD using its context and deserializer
>>> vertices = VertexRDD(rdd)
>>> sorted(vertices.mapValues(lambda x: (x + ":" + x)).collect())
[(1, 'a:a'), (2, 'b:b'), (3, 'c:c')]
```

mapVertexPartitions(f)

The method enables function “f” to be executed on every partition of the vertex RDD.

innerJoin(other)

Performs an intersection operation on two vertex RDD objects. The properties of vertices with same id in the current RDD and other RDD are combined into a tuple.

filter(f)

The filter operation conditionally selects a subset of vertices from the given vertex RDD object. The function "f" specifies the lambda condition function.

leftJoin(f)

This method is a left outer join operation on two vertex RDD objects. The properties of the common vertices in the two RDDs are combined into a tuple.

The actions on VertexRDDs are similar to the ones for PySpark RDDS. For example, a collect() operation converts the content of the entire vertex RDD into an array:

```
>>> rdd0 = sc.parallelize([(1, ("Alice", "manager")), (2, ("Bob", "engineer"))])
>>> vertices = VertexRDD(rdd0)
>>> vertices.collect()
>>> [(1, ("Alice", "manager")), (2, ("Bob", "engineer"))]
```

Note: PySpark, the Python bindings for Apache Spark uses different tricks to access and manipulate data between Python client and the backend JVM-based Scala objects. First, it uses Py4J library to access the JVM objects. Second, in case of actions it uses file serialization and deserialization to avoid expensive data transfer between JVM and Python clients. The same methodology is followed in VertexRDD objects. A transformation operation creates simultaneously creates a PipelinedVertexRDD object in Python and a PythonVertexRDD object in Scala. The PythonVertexRDD object sees the RDDs as bytearrays. This object is responsible to spawn Python workers to eventually perform an operation. The object diagram is shown in Figure 1.

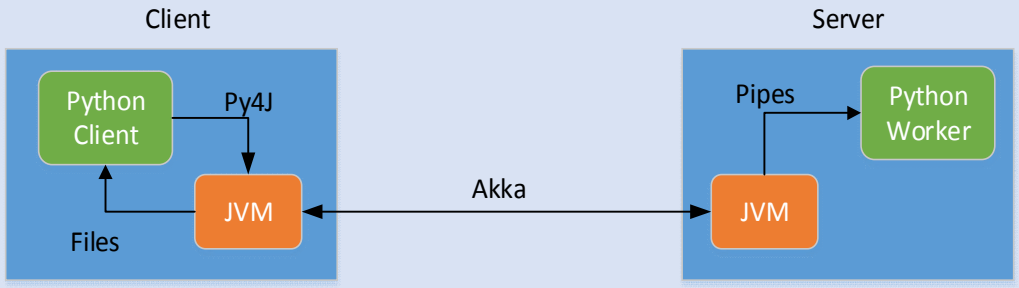

Figure 1: Method of operation of PySpark and PyGraphX

An example object relationship is shown in Figure 2.

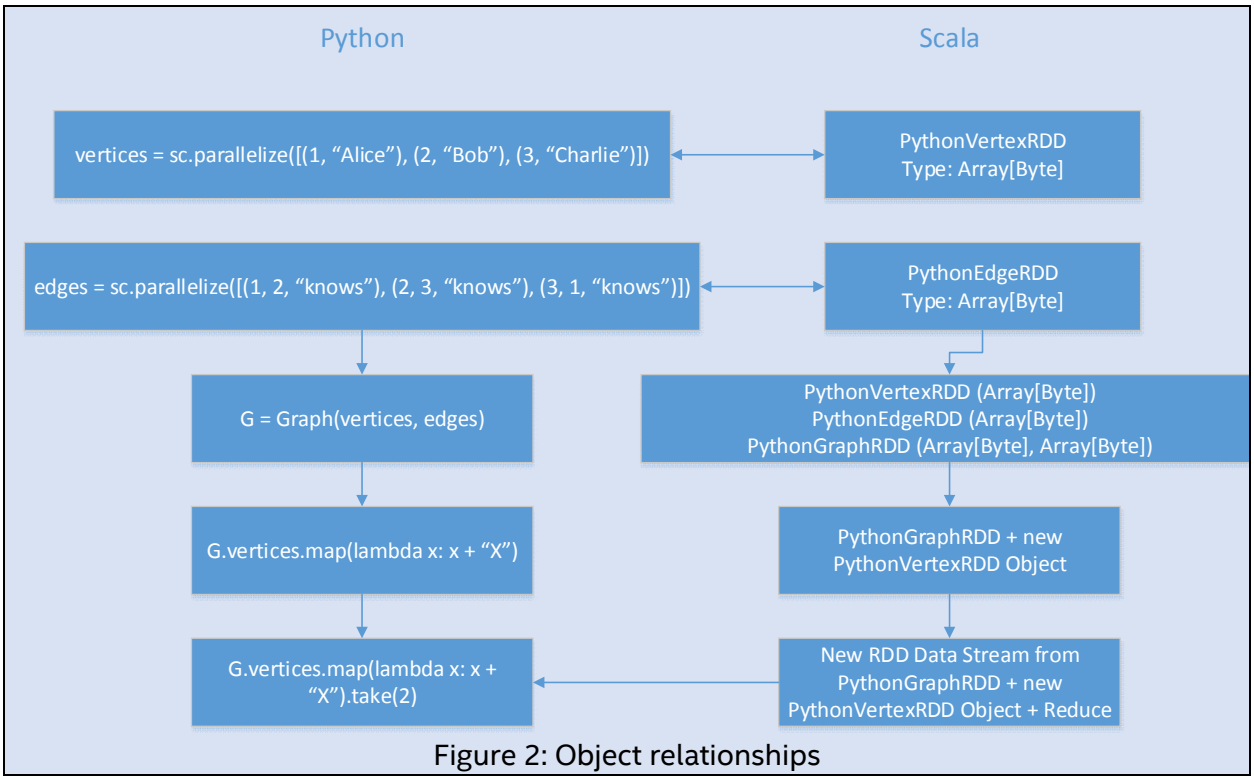

Figure 2: Object relationships

EdgeRDD Objects

EdgeRDD class is provided to support transformation and action operations on a distributed list of edges of a graph. For example, an edges class with a string values as property can be used as:

```
>>> rdd0 = sc.parallelize([(1, 3, (“worksWith”)), (2,3,  (“knows”)), (3, 1, (“likes”))])
>>> edges = EdgeRDD(rdd0)
>>> edges.count()
>>> 3
```

class EdgeRDD(parent-RDD, spark-context, de-serializer):

An EdgeRDD is an unordered indexed RDD of edge tuples. The edge tuples are made up of source vertex ids, destination vertex ids (both of data type long) and edge property object which can be of any standard Python data type. Edges are created from parent RDDs as following:

```
>>> rdd = sc.parallelize([(1, 3, (“worksWith”)), (2, 3, (“knows”)), (3, 1, (“likes”)), (4, ,1, (“knows”))])
>>> # Create from a parent RDD using its context and deserializer
>>> edges = EdgeRDD(rdd)
>>> # Create from a parent RDD using a separate context
>>> edges = EdgeRDD(rdd, sc)
>>> # Create from a parent RDD using separate context and deserializer
```

```
>>> edges = EdgeRDD(rdd, sc, BatchSerializer(CloudPickleSerializer()))
```

EdgeRDD provides the following additional methods:

mapValues(f)

Map only the edge properties of a EdgeRDD. Keys remain unchanged. The argument "f" is a lambda function to be executed for every edge in the RDD.

```
>>> rdd = sc.parallelize([(1, 2, "worksWith"), (2, 3, "knows"), (3, 2, "likes")])
>>> # Create from a parent RDD using its context and deserializer
>>> edges = EdgeRDD(rdd)
>>> sorted(edges.mapValues(lambda x: (x + ":" + x)).collect())
[(1, 2, "worksWith:worksWith"), (2, 3, "knows:knows"), (3, 2, 'likes:likes')]
```

innerJoin(other)

Performs an intersection operation on two edge RDD objects. The properties of edge with source and destination vertex id pair in the current RDD and other RDD are combined into a tuple.

filter(f)

The filter operation conditionally selects a subset of edges from the given edge RDD object. The function "f" specifies the lambda condition function.

leftJoin(f)

The method implements the left outer join operation on two edge RDD objects. The properties of the common edges (with same source and destination vertex ids) in the two RDDs are combined into a tuple.

The actions on EdgeRDDs are similar to the ones for PySpark RDDS. For example:

```
>>> rdd = sc.parallelize([(1, 2, ("worksWith")), (2, 3, ("knows"))])
>>> edges = EdgeRDD(rdd)
>>> edges.collect()
>>> [(1, 2, ("worksWith")), (2, 3, ("knows"))]
```

Note: The underlying architecture of EdgeRDD is very similar to VertexRDD. The corresponding objects are PythonEdgeRDD in Scala.

GraphRDD Objects

The GraphRDD data type provides graph algorithms on vertex and edge RDD objects. It also provides a *Pregel* API so that users can write their own graph algorithm using vertex programs.

class GraphRDD(vertexRDD, edgeRDD):

The GraphRDD object is created from vertex and edge RDDs. It inherits the data types of vertex/edge properties, Spark context and deserializer from the input vertex and edge RDD objects. For example:

```
>>> rdd0 = sc.parallelize([(1, (“Alice”, “manager”)), (2, (“Bob”, “engineer”))])
>>> vertices = VertexRDD(rdd0)
>>> rdd1 = sc.parallelize([(1, 2, (“worksWith”)), (2, 3, (“knows”))])
>>> edges = EdgeRDD(rdd1)
>>> graph = GraphRDD(vertices, edges)
```

GraphRDD provides the following methods.

fromEdges(edgeRDD)

This method creates a graph from a given edgeRDD. In this case vertices are created only from vertex ids with null properties.

mapEdges(f)

Allows to run a lambda function “f” in all edges in the graph.

mapVertices()

Allows to run a lambda function “f” in all vertices in the graph.

reverse()

Reverse all the edges in the graph

subgraph(f)

Extract a subgraph which is of type GraphRDD according to the filtering function “f”

groupEdges(f)

Merge the edges of the graph with the same source and destination vertex ids into a single edge. The function “f” is the merge function

joinVertices(vertexRDD)

This is an innerJoin function on the vertices of the given graph with another vertex RDD object.

outerJoinVertices(vertexRDD)

Calculate the union operation on all the vertices in the graph with the given vertex RDD object.

pagerank()

Calculate the PageRank algorithm on all the vertices of the graph

trianglecount()

Measure the number of triangles in the graph

Pregel(initial_message, vertex_program, send_message, combine_message)

Specify the vertexProgram function to execute on every vertex of the graph. The Pregel API enables an iterative computation of the vertexProgram till there are no messages exchanged in the graph.

Dependency

Java API Implementation

PyGraphX bindings depends on the Java GraphX API. Similar to Python, the Java API also contains JavaVertexRDD, JavaEdgeRDD and JavaGraph classes – the Java bindings of the Vertex, Edge and Graph methods defined in Scala.

class JavaVertexRDD[VD](val vertexRDD: VertexRDD[VD]) extends JavaPairRDD[JLong, VD]

JavaVertexRDD extends JavaPairRDD comprising of a tuple pair of VertexID as JLong and the Vertex Data.

mapValues[VD2](f: JFunction[VD, VD2]): JavaVertexRDD[VD2]

Execute the function f on the VertexData VD and return result of type VD2

diff(other: JavaVertexRDD[VD]): JavaVertexRDD[VD]

Returns the difference between 2 JavaVertexRDDs

innerZipJoin(other: JavaVertexRDD[U], f: JFunction3[JLong, VD, U, VD2]): JavaVertexRDD[VD2]

Inner zip join 2 JavaVertexRDD sharing the same index

innerJoin(other: JavaPairRDD[JLong, U], f: JFunction3[JLong, VD, U, VD2]): JavaVertexRDD[VD2]

Inner join rdd with a Java Pair RDD consisting of attribute pairs

leftZipJoin(other: JavaVertexRDD[U], f: JFunction3[JLong, VD,Optional[U], VD2]): JavaVertexRDD[VD2]

Left zip join 2 JavaVertexRDD sharing the same index

leftJoin(other: JavaPairRDD[JLong, U], f: JFunction3[JLong, VD, Optional[U], VD2]): JavaVertexRDD[VD2]

Left rdd with a Java Pair RDD consisting of attribute pairs

aggregateWithIndex(messages: JavaPairRDD[JLong, VD2], reduceFunc: JFunction2[VD2, VD2, VD2]): JavaVertexRDD[VD2]

Aggregate vertices in messages which share the same vertex id using given reduceFunc returning a JavaVertexRDD

class JavaEdgeRDD[ED, VD](override val rdd: EdgeRDD[ED]) extends JavaRDD[Edge[ED]](rdd)

JavaEdgeRDD extends JavaRDD[Edge[ED]] type and takes an EdgeRDD. The interface is simply extending the scala api to java here.

mapValues[ED2](f: JFunction[Edge[ED], ED2]): JavaEdgeRDD[ED2, VD]

Execute function f on edges of type ED and outputs type ED2 and return a JavaEdgeRDD

reverse()

Reverses the edges

filter(epred: JFunction[EdgeTriplet[VD,ED], JBoolean], vpred: JFunction2[JLong, VD, JBoolean]): JavaEdgeRDD[ED, VD]

Filters the edges based on the edge and vertex predicates

innerJoin(other: JavaEdgeRDD[ED2, _], f: JFunction4[JLong, JLong, ED, ED2, ED3]): JavaEdgeRDD[ED3, ED]

Inner join 2 edge rdd given the join function f

API Implementation to support Py4J

Note that VertexRDDs in GraphX are a collection of tuples of vertex id of type long and vertex attributes (user-defined class). This structure is analogous to a key-value pair and can be efficiently handled by an RDD designed for pairwise operations. We leverage the existing PairWiseRDD and JavaPairRDD method definitions. Hence we define a new PythonPairRDD class in Scala.

class PythonPairRDD(prev: RDD[Array[Byte]]) extends RDD[(Long, Array[Byte])](prev)

compute()

Deserializes the python byte array to extract the vertex ID and create a tuple of VertexId: Long and the remaining python byte array data

Next we leverage the PythonPairRDD implementation along with JavaVertexRDD to define PythonVertexRDD. PythonVertexRDD takes a PythonRDD as an argument and extends JavaVertexRDD. Internally, the PythonRDD is first converted into a PythonPairRDD where the vertex Id is deserialized to create a tuple of vertex id and python byte array data. We leverage the PythonRDD compute mechanism to execute user defined functions as shown in Figure 1.

```
class PythonVertexRDD(rdd: PythonRDD) extends
JavaVertexRDD[Array[Byte]](VertexRDD(new PythonPairRDD(rdd)))
```

Similar to PythonPairRDD, we define a triplet class for Edge implementation which the PythonEdgeRDD will extend. This RDD will deserialize and extract the source and target information from the python bytearray hence converting a byte array to an Edge[Array[Byte]]

```
class PythonTripletRDD(prev: RDD[Array[Byte]]) extends RDD[(Long, Long,
Array[Byte]](prev)
```

Our PythonEdgeRDD is next defined as follows.

class PythonEdgeRDD(rdd: PythonRDD) extends
JavaEdgeRDD[Array[Byte]](EdgeRDD.fromEdges(new PythonTripletRDD(rdd)))

PythonEdgeRDD extends the JavaEdgeRDD and consumes a PythonRDD. Internally it deserializes the source and edge vertex IDs and initializes the underying JavaEdgeRDD. The remaining data is kept as byte array and we leverage existing PythonRDD functionality to execute user defined functions through PythonRDD compute mechanism as mentioned in Figure 1. PythonGraph need to be implemented by leveraging the existing partitioning strategies and APIs implemented by the VertexRDD, EdgeRDD and Graph interfaces. PythonGraph extends the JavaGraph class and can be created from PythonVertexRDD and PythonEdgeRDD.

```
class PythonGraph (
  @transient val vertexRDD: PythonVertexRDD,
  @transient val edgeRDD: PythonEdgeRDD)
extends JavaGraph[Array[Byte], Array[Byte]] (JavaGraph.create(vertexRDD, edgeRDD, null))
```

def persist(newLevel: StorageLevel = StorageLevel.MEMORY_ONLY): PythonGraph

Persist this RDD with the default storage level (MEMORY_ONLY)

def cache(): PythonGraph

Persist this RDD with the default storage level (MEMORY_ONLY)

def checkpoint(): Unit

Mark this Graph for checkpointing. It will be saved to a file inside the checkpoint directory set with **SparkContext.setCheckpointDir()** and all references to its parent will be removed. This function must be called before any job has been executed on this Graph. It is strongly recommended that this Graph is persisted in memory, otherwise saving it on a file will require recomputation.

def isCheckpointed: Boolean

Return whether this Graph has been checkpointed or not.

def getCheckpointFiles: Seq[String]

Gets the name of the files to which this Graph was checkpointed.

def unpersist(blocking: Boolean = true): PythonGraph

Uncaches both vertices and edges of this graph. This is useful in iterative algorithms that

build a new graph in each iteration.

def partitionBy(partitionStrategy: PartitionStrategy): PythonGraph

Repartitions the edges in the graph according to `partitionStrategy` which is to be used when partitioning the edges in the graph.

def partitionBy(partitionStrategy: PartitionStrategy, numPartitions: Int): PythonGraph

Repartitions the edges in the graph according to `partitionStrategy` to number of edge partitions in the new graph

def mapVertices(map: (VertexId, Array[Byte])=> Array[Byte]): PythonGraph

Transforms each vertex attribute in the graph using the map function.

def mapEdges (map: Edge[Array[Byte]] => Array[Byte]): PythonGraph

Transforms each edge attribute in the graph using the map function

def mapEdges (map: (PartitionID, Iterator[Edge[Array[Byte]]]) => Iterator[Array[Byte]]) : PythonGraph

Transforms each edge attribute using the map function, passing it a whole partition at a time

def reverse: PythonGraph

Reverses all edges in the graph

def subgraph(epred: EdgeTriplet[Array[Byte],Array[Byte]] => Boolean = (x => true), vpred: (Long, Array[Byte]) => Boolean = ((v, d) => true)) : PythonGraph

Restricts the graph to only the vertices and edges satisfying the predicates

def mask(other: PythonGraph): PythonGraph

Restricts the graph to only the vertices and edges that are also in `other`, but keeps the attributes from this graph.

def groupEdges(merge: (Array[Byte], Array[Byte]) => Array[Byte]): PythonGraph

Merges multiple edges between two vertices into a single edge. For correct results, the graph must have been partitioned using same [[partitionBy]].

def aggregateMessages(

sendMsg: EdgeContext[Array[Byte], Array[Byte], Array[Byte]] => Unit,

mergeMsg: (Array[Byte], Array[Byte]) => Array[Byte],

tripletFields: TripletFields = TripletFields.All)

: PythonVertexRDD

Aggregates values from the neighboring edges and vertices of each vertex.

def outerJoinVertices (other: PythonVertexRDD) (mapFunc: (VertexId, Array[Byte], Option[Array[Byte]]) => Array[Byte]) : PythonGraph

Joins the vertices with vertices in other and merges the results using `mapFunc`

Partitioning Strategy

Py4J Interface relies on the JavaVertexRDD and JavaEdgeRDD to leverage the partitioning strategies. As the Vertex IDs are deserialized from the byte array and are native Java Longs, these can be then used in all methods available with VertexRDD and EdgeRDD to implement the partitioning logic based on src ids and dest ids.

In Python – the partitionBy can be called with the following definition which will invoke the Py4j implementation of partitionBy in PythonGraph.

graph.partitionBy(*numPartitions*, *partitionFunc=<function portable_hash at 0x7f8d1be68a28>*)¶