Guía de programación de GraphX

Descripción general

GraphX es un nuevo componente de Spark para grafos y computación paralela a grafos. A grandes rasgos, GraphX amplía el RDD de Spark al introducir una nueva abstracción de grafos : un multigrafo dirigido con propiedades asociadas a cada vértice y arista. Para facilitar la computación de grafos, GraphX expone un conjunto de operadores fundamentales (p. ej., subgrafo , joinVertices y addedMessages), así como una versión optimizada de la API de Pregel . Además, GraphX incluye una colección creciente de algoritmos y constructores de grafos para simplificar las tareas de análisis de grafos.

Empezando

Para comenzar, primero debes importar Spark y GraphX a tu proyecto, de la siguiente manera:

```
import org.apache.spark._
```

```
import org.apache.spark.graphx._
// To make some of the examples work we will also need RDD
import org.apache.spark.rdd.RDD
```

Si no usa el shell de Spark, también necesitará un archivo `SparkContext`. Para obtener más información sobre cómo comenzar a usar Spark, consulte la Guía de inicio rápido de Spark .

El gráfico de propiedades

El grafo de propiedades es un multigrafo dirigido con objetos definidos por el usuario asociados a cada vértice y arista. Un multigrafo dirigido es un grafo dirigido con múltiples aristas paralelas que comparten el mismo vértice de origen y destino. La compatibilidad con aristas paralelas simplifica el modelado en escenarios donde pueden existir múltiples relaciones (p. ej., compañero de trabajo y amigo) entre los mismos vértices. Cada vértice está codificado por un identificador *único* de 64 bits (`VertexId`). GraphX no impone restricciones de ordenación a los identificadores de vértice. De igual forma, las aristas tienen sus correspondientes identificadores de vértice de origen y destino.

El grafo de propiedades se parametriza sobre los tipos de vértice (`VD`) y arista (`ED`). Estos son los tipos de objetos asociados a cada vértice y arista, respectivamente.

GraphX optimiza la representación de tipos de vértices y aristas cuando son tipos de datos primitivos (por ejemplo, int, double, etc.), reduciendo el espacio en memoria al almacenarlos en matrices especializadas.

En algunos casos, puede ser conveniente tener vértices con diferentes tipos de propiedades en el mismo grafo. Esto se puede lograr mediante herencia. Por ejemplo, para modelar usuarios y productos como un grafo bipartito, podríamos hacer lo siguiente:

```
class VertexProperty()
case class UserProperty(val name: String) extends VertexProperty
case class ProductProperty(val name: String, val price: Double) extends
VertexProperty
// The graph might then have the type:
var graph: Graph[VertexProperty, String] = null
```

Al igual que los RDD, los grafos de propiedades son inmutables, distribuidos y tolerantes a fallos. Los cambios en los valores o la estructura del grafo se logran generando un nuevo grafo con los cambios deseados. Cabe destacar que partes sustanciales del grafo original (es decir, la estructura, los atributos y los índices intactos) se reutilizan en el nuevo grafo, lo que reduce el costo de esta estructura de datos inherentemente funcional. El grafo se particiona entre los ejecutores mediante diversas heurísticas de partición de vértices. Al igual que con los RDD, cada partición del grafo puede recrearse en una máquina diferente en caso de fallo.

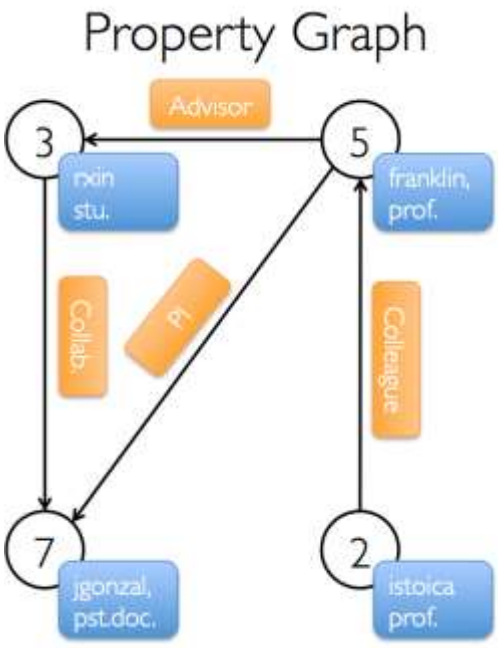
Lógicamente, el grafo de propiedades corresponde a un par de colecciones tipificadas (RDD) que codifican las propiedades de cada vértice y arista. En consecuencia, la clase de grafo contiene miembros para acceder a los vértices y aristas del grafo:

```
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```

Las clases `VertexRDD[VD]` y `EdgeRDD[ED]` extienden y son versiones optimizadas de `RDD[(VertexId, VD)]` y `RDD[Edge[ED]]` respectivamente. Ambas `VertexRDD[VD]` y `EdgeRDD[ED]` proporcionan funcionalidad adicional basada en el cálculo de grafos y aprovechan las optimizaciones internas. Analizamos las API `VertexRDD` VertexRDD y `EdgeRDD` EdgeRDD con más detalle en la sección sobre RDD de vértices y aristas, pero por ahora pueden considerarse simplemente RDD de la forma: `RDD[(VertexId, VD)]` y `RDD[Edge[ED]]`.

Ejemplo de gráfico de propiedades

Supongamos que queremos construir un grafo de propiedades que conste de los distintos colaboradores del proyecto GraphX. La propiedad vértice podría contener el nombre de usuario y la ocupación. Podríamos anotar las aristas con una cadena que describa las relaciones entre los colaboradores:



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

El gráfico resultante tendría la firma de tipo:

```
val userGraph: Graph[(String, String), String]
```

Existen numerosas maneras de construir un grafo de propiedades a partir de archivos sin procesar, RDD e incluso generadores sintéticos, y se describen con más detalle en la sección sobre constructores de grafos . Probablemente, el método más general sea usar el objeto Graph . Por ejemplo, el siguiente código construye un grafo a partir de una colección de RDD:

```
// Assume the SparkContext has already been constructed
```

```
val sc: SparkContext
// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(Seq((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
                     (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))
// Create an RDD for edges
val relationships: RDD[Edge[String]] =
  sc.parallelize(Seq(Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
                     Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))
// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)
```

En el ejemplo anterior, utilizamos la `Edge`clase case. Las aristas tienen a `srcId`y a, `dstId`que corresponden a los identificadores de vértice de origen y destino. Además, la `Edge` clase tiene un `attr`miembro que almacena la propiedad de la arista.

Podemos deconstruir un gráfico en las respectivas vistas de vértice y arista utilizando los miembros `graph.vertices` y respectivamente.`graph.edges`

```
val graph: Graph[(String, String), String] // Constructed from above
// Count all users which are postdocs
graph.vertices.filter { case (id, (name, pos)) => pos == "postdoc" }.count
// Count all the edges where src > dst
graph.edges.filter(e => e.srcId > e.dstId).count
```

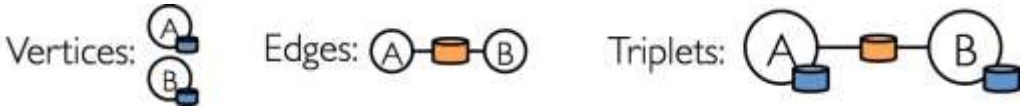
Tenga en cuenta que `graph.vertices`devuelve un `VertexRDD[(String, String)]`que extiende `RDD[(VertexId, (String, String))]`, por lo que usamos la `case`expresión de Scala para deconstruir la tupla. Por otro lado, `graph.edges`devuelve un `EdgeRDD`que contiene `Edge[String]`objetos. También podríamos haber usado el constructor de tipo de clase case como se muestra a continuación:

```
graph.edges.filter { case Edge(src, dst, prop) => src > dst }.count
```

Además de las vistas de vértices y aristas del grafo de propiedades, GraphX también expone una vista de tripletes. Esta vista une lógicamente las propiedades de vértices y aristas, generando `RDD[EdgeTriplet[VD, ED]]`instancias contenedoras de la `EdgeTriplet`clase. Esta *unión* se puede expresar con la siguiente expresión SQL:

```
SELECT src.id, dst.id, src.attr, e.attr, dst.attr
FROM edges AS e LEFT JOIN vertices AS src, vertices AS dst
ON e.srcId = src.Id AND e.dstId = dst.Id
```

o gráficamente como:



La `EdgeTriplet`clase extiende la `Edge`clase añadiendo los miembros `` `srcAttr` `` y `dstAttr``, que contienen las propiedades de origen y destino, respectivamente. Podemos usar la vista de tripletes de un grafo para representar una colección de cadenas que describen las relaciones entre usuarios.

```
val graph: Graph[(String, String), String] // Constructed from above
// Use the triplets view to create an RDD of facts.
val facts: RDD[String] =
  graph.triplets.map(triplet =>
    triplet.srcAttr._1 + " is the " + triplet.attr + " of " +
triplet.dstAttr._1)
facts.collect.foreach(println(_))
```

Operadores gráficos

Así como los RDD tienen operaciones básicas como `map`, `filter`, y `reduceByKey`, los grafos de propiedades también tienen un conjunto de operadores básicos que toman funciones definidas por el usuario y generan nuevos grafos con propiedades y estructura transformadas. Los operadores principales con implementaciones optimizadas se definen en , `Graph`y los operadores convenientes, expresados como composiciones de los operadores principales, se definen en `GraphOps`. Sin embargo, gracias a las funciones implícitas de Scala, los operadores en `GraphOps`están disponibles automáticamente como miembros de `Graph`. Por ejemplo, podemos calcular el grado de entrada de cada vértice (definido en `GraphOps`) mediante la siguiente fórmula:

```
val graph: Graph[(String, String), String]
// Use the implicit GraphOps.inDegrees operator
val inDegrees: VertexRDD[Int] = graph.inDegrees
```

La razón para diferenciar entre las operaciones principales de grafos `GraphOps`es poder soportar diferentes representaciones gráficas en el futuro. Cada representación gráfica debe proporcionar implementaciones de las operaciones principales y reutilizar muchas de las operaciones útiles definidas en `GraphOps`.

Lista resumida de operadores

A continuación, se presenta un breve resumen de la funcionalidad definida en ambos `Graph`, `GraphOps`pero presentada como miembros de Graph para simplificar. Tenga en cuenta que se han simplificado algunas firmas de funciones (p. ej., se han eliminado los argumentos predeterminados y las restricciones de tipo) y se han eliminado algunas funciones más avanzadas. Por lo tanto, consulte la documentación de la API para obtener la lista oficial de operaciones.

```
/** Summary of the functionality in the property graph */
class Graph[VD, ED] {
```

```
  // Information about the Graph
==================================================================
  val numEdges: Long
  val numVertices: Long
  val inDegrees: VertexRDD[Int]
  val outDegrees: VertexRDD[Int]
  val degrees: VertexRDD[Int]
  // Views of the graph as collections
==================================================================
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
  val triplets: RDD[EdgeTriplet[VD, ED]]
  // Functions for caching graphs
==================================================================
  def persist(newLevel: StorageLevel = StorageLevel.MEMORY_ONLY): Graph[VD, ED]
  def cache(): Graph[VD, ED]
  def unpersistVertices(blocking: Boolean = false): Graph[VD, ED]
  // Change the partitioning heuristic
==================================================================
  def partitionBy(partitionStrategy: PartitionStrategy): Graph[VD, ED]
  // Transform vertex and edge attributes
==================================================================
  def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]
  def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
  def mapEdges[ED2](map: (PartitionID, Iterator[Edge[ED]]) => Iterator[ED2]):
Graph[VD, ED2]
  def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
  def mapTriplets[ED2](map: (PartitionID, Iterator[EdgeTriplet[VD, ED]]) =>
Iterator[ED2])
    : Graph[VD, ED2]
  // Modify the graph structure
==================================================================
  def reverse: Graph[VD, ED]
  def subgraph(
      epred: EdgeTriplet[VD,ED] => Boolean = (x => true),
      vpred: (VertexId, VD) => Boolean = ((v, d) => true))
    : Graph[VD, ED]
  def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
  def groupEdges(merge: (ED, ED) => ED): Graph[VD, ED]
  // Join RDDs with the graph
==================================================================
  def joinVertices[U](table: RDD[(VertexId, U)])(mapFunc: (VertexId, VD, U) =>
VD): Graph[VD, ED]
  def outerJoinVertices[U, VD2](other: RDD[(VertexId, U)])
```

```
      (mapFunc: (VertexId, VD, Option[U]) => VD2)
    : Graph[VD2, ED]
  // Aggregate information about adjacent triplets
============================================
  def collectNeighborIds(edgeDirection: EdgeDirection):
VertexRDD[Array[VertexId]]
  def collectNeighbors(edgeDirection: EdgeDirection): VertexRDD[Array[(VertexId,
VD)]]
  def aggregateMessages[Msg: ClassTag](
      sendMsg: EdgeContext[VD, ED, Msg] => Unit,
      mergeMsg: (Msg, Msg) => Msg,
      tripletFields: TripletFields = TripletFields.All)
    : VertexRDD[A]
  // Iterative graph-parallel computation
============================================================
  def pregel[A](initialMsg: A, maxIterations: Int, activeDirection:
EdgeDirection)(
      vprog: (VertexId, VD, A) => VD,
      sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
      mergeMsg: (A, A) => A)
    : Graph[VD, ED]
  // Basic graph algorithms
========================================================================
  def pageRank(tol: Double, resetProb: Double = 0.15): Graph[Double, Double]
  def connectedComponents(): Graph[VertexId, ED]
  def triangleCount(): Graph[Int, ED]
  def stronglyConnectedComponents(numIter: Int): Graph[VertexId, ED]
}
```

Operadores de propiedades

Al igual que el `map`operador RDD, el gráfico de propiedades contiene lo siguiente:

```
class Graph[VD, ED] {
  def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]
  def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
  def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
}
```

Cada uno de estos operadores produce un nuevo gráfico con las propiedades de vértice o arista modificadas por la `map`función definida por el usuario.

Tenga en cuenta que, en ningún caso, la estructura del grafo se ve afectada. Esta es una característica clave de estos operadores, que permite que el grafo resultante reutilice los índices estructurales del grafo original. Los siguientes fragmentos son lógicamente

equivalentes, pero el primero no conserva los índices estructurales y no se beneficiaría de las optimizaciones del sistema GraphX:

```
val newVertices = graph.vertices.map { case (id, attr) => (id, mapUdf(id, attr))
}
val newGraph = Graph(newVertices, graph.edges)
```

En su lugar, utilice `mapVertices`para conservar los índices:

```
val newGraph = graph.mapVertices((id, attr) => mapUdf(id, attr))
```

Estos operadores se utilizan a menudo para inicializar el grafo para un cálculo específico o para descartar propiedades innecesarias. Por ejemplo, dado un grafo con los grados de salida como propiedades del vértice (más adelante describiremos cómo construir dicho grafo), lo inicializamos para PageRank:

```
// Given a graph where the vertex property is the out degree
val inputGraph: Graph[Int, String] =
  graph.outerJoinVertices(graph.outDegrees)((vid, _, degOpt) =>
degOpt.getOrElse(0))
// Construct a graph where each edge contains the weight
// and each vertex is the initial PageRank
val outputGraph: Graph[Double, Double] =
  inputGraph.mapTriplets(triplet => 1.0 / triplet.srcAttr).mapVertices((id, _) =
> 1.0)
```

Operadores estructurales

Actualmente, GraphX solo admite un conjunto simple de operadores estructurales de uso común y esperamos añadir más en el futuro. A continuación, se presenta una lista de los operadores estructurales básicos.

```
class Graph[VD, ED] {
  def reverse: Graph[VD, ED]
  def subgraph(epred: EdgeTriplet[VD,ED] => Boolean,
               vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
  def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
  def groupEdges(merge: (ED, ED) => ED): Graph[VD,ED]
}
```

El `reverse`operador devuelve un nuevo grafo con todas las direcciones de las aristas invertidas. Esto puede ser útil, por ejemplo, al intentar calcular el PageRank inverso. Dado que la operación inversa no modifica las propiedades de los vértices ni las aristas, ni cambia el número de aristas, se puede implementar eficientemente sin mover ni duplicar datos.

El `subgraph`operador toma predicados de vértice y arista y devuelve el grafo que contiene únicamente los vértices que satisfacen el predicado de vértice (evalúan como verdadero) y

las aristas que satisfacen el predicado de arista *y conectan los vértices que satisfacen el predicado de vértice* . El `subgraph` operador puede utilizarse en diversas situaciones para restringir el grafo a los vértices y aristas de interés o para eliminar enlaces rotos. Por ejemplo, en el siguiente código eliminamos los enlaces rotos:

```
// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(Seq((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
                       (5L, ("franklin", "prof")), (2L, ("istoica", "prof")),
                       (4L, ("peter", "student"))))
// Create an RDD for edges
val relationships: RDD[Edge[String]] =
  sc.parallelize(Seq(Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
                       Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi"),
                       Edge(4L, 0L, "student"),   Edge(5L, 0L, "colleague")))
// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)
// Notice that there is a user 0 (for which we have no information) connected to
users
// 4 (peter) and 5 (franklin).
graph.triplets.map(
  triplet => triplet.srcAttr._1 + " is the " + triplet.attr + " of " +
triplet.dstAttr._1
).collect.foreach(println(_))
// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")
// The valid subgraph will disconnect users 4 and 5 by removing user 0
validGraph.vertices.collect.foreach(println(_))
validGraph.triplets.map(
  triplet => triplet.srcAttr._1 + " is the " + triplet.attr + " of " +
triplet.dstAttr._1
).collect.foreach(println(_))
```

Tenga en cuenta que en el ejemplo anterior solo se proporciona el predicado de vértice. El `subgraph`operador predeterminado es `true`si no se proporcionan los predicados de vértice o arista.

El `mask`operador construye un subgrafo devolviendo un grafo que contiene los vértices y las aristas que también se encuentran en el grafo de entrada. Esto puede usarse junto con el `subgraph`operador para restringir un grafo según las propiedades de otro grafo relacionado. Por ejemplo, podríamos ejecutar componentes conexos utilizando el grafo con vértices faltantes y luego restringir la respuesta al subgrafo válido.

```
// Run Connected Components
val ccGraph = graph.connectedComponents() // No longer contains missing field
// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")
// Restrict the answer to the valid subgraph
val validCCGraph = ccGraph.mask(validGraph)
```

El `groupEdges`operador fusiona aristas paralelas (es decir, aristas duplicadas entre pares de vértices) en el multigrafo. En muchas aplicaciones numéricas, es posible *sumar* aristas paralelas (combinando sus pesos) en una sola arista, reduciendo así el tamaño del grafo.

Unirse a los operadores

En muchos casos es necesario unir datos de colecciones externas (RDD) con grafos. Por ejemplo, podríamos tener propiedades de usuario adicionales que queramos fusionar con un grafo existente o podríamos querer extraer propiedades de vértices de un grafo a otro. Estas tareas se pueden realizar mediante operadores *de unión* . A continuación, se enumeran los principales operadores de unión:

```
class Graph[VD, ED] {
  def joinVertices[U](table: RDD[(VertexId, U)])(map: (VertexId, VD, U) => VD)
    : Graph[VD, ED]
  def outerJoinVertices[U, VD2](table: RDD[(VertexId, U)])(map: (VertexId, VD,
Option[U]) => VD2)
    : Graph[VD2, ED]
}
```

El `joinVertices`operador une los vértices con el RDD de entrada y devuelve un nuevo grafo con las propiedades de los vértices obtenidas al aplicar la `map`función definida por el usuario al resultado de los vértices unidos. Los vértices sin un valor coincidente en el RDD conservan su valor original.

Tenga en cuenta que si el RDD contiene más de un valor para un vértice dado, solo se usará uno. Por lo tanto, se recomienda que el RDD de entrada sea único mediante la siguiente instrucción, que también *preindexará* los valores resultantes para agilizar considerablemente la unión posterior.

```
val nonUniqueCosts: RDD[(VertexId, Double)]
val uniqueCosts: VertexRDD[Double] =
  graph.vertices.aggregateUsingIndex(nonUnique, (a,b) => a + b)
val joinedGraph = graph.joinVertices(uniqueCosts)(
  (id, oldCost, extraCost) => oldCost + extraCost)
```

La función más general `outerJoinVertices`se comporta de forma similar, `joinVertices` excepto que la `map`función definida por el usuario se aplica a todos los vértices y puede cambiar el tipo de propiedad del vértice. Dado que no todos los vértices pueden tener un valor coincidente en el RDD de entrada, la `map` función toma

un `Option`tipo. Por ejemplo, podemos configurar un grafo para PageRank inicializando las propiedades del vértice con su [nombre del vértice `outDegree`].

```
val outDegrees: VertexRDD[Int] = graph.outDegrees
val degreeGraph = graph.outerJoinVertices(outDegrees) { (id, oldAttr, outDegOpt)
=>
  outDegOpt match {
    case Some(outDeg) => outDeg
    case None => 0 // No outDegree means zero outDegree
  }
}
```

Quizás haya observado el `f(a)(b)`patrón de función currificada con listas de parámetros múltiples (p. ej.,) utilizado en los ejemplos anteriores. Si bien podríamos haber escrito esto también `f(a)(b)`como `f(a,b)`, significaría que la inferencia de tipos en bno dependería de a. En consecuencia, el usuario tendría que proporcionar una anotación de tipo para la función definida por el usuario:

```
val joinedGraph = graph.joinVertices(uniqueCosts,
  (id: VertexId, oldCost: Double, extraCost: Double) => oldCost + extraCost)
```

Agregación de vecindarios

Un paso clave en muchas tareas de análisis de grafos es agregar información sobre la vecindad de cada vértice. Por ejemplo, podríamos querer saber el número de seguidores de cada usuario o su edad promedio. Muchos algoritmos iterativos de grafos (p. ej., PageRank, Ruta más corta y componentes conectados) agregan repetidamente las propiedades de los vértices vecinos (p. ej., el valor actual de PageRank, la ruta más corta al origen y el identificador de vértice más pequeño alcanzable).

Para mejorar el rendimiento, el operador de agregación principal cambió de [nombre del operador] `graph.mapReduceTripletsal` nuevo [nombre del operador] `graph.AggregateMessages`. Si bien los cambios en la API son relativamente pequeños, a continuación ofrecemos una guía de transición.

Mensajes agregados (agregateMessages)

La operación principal de agregación en GraphX es `aggregateMessages`. Este operador aplica una `sendMsg`función definida por el usuario a cada *triplete de aristas* del grafo y luego utiliza dicha `mergeMsg`función para agregar esos mensajes en su vértice de destino.

```
class Graph[VD, ED] {
  def aggregateMessages[Msg: ClassTag](
      sendMsg: EdgeContext[VD, ED, Msg] => Unit,
      mergeMsg: (Msg, Msg) => Msg,
      tripletFields: TripletFields = TripletFields.All)
    : VertexRDD[Msg]
```

```
}
```

La `sendMsg`función definida por el usuario toma un `EdgeContext`, que expone los atributos de origen y destino junto con el atributo de borde y las funciones (`sendToSrc`, y `sendToDst`) para enviar mensajes a dichos atributos. Es similar `sendMsg`a la función *map*`mergeMsg` de map-reduce. Esta función definida por el usuario toma dos mensajes destinados al mismo vértice y genera un único mensaje. Es similar `mergeMsg`a la función *reduce* de map-reduce. El `aggregateMessages`operador devuelve un , `VertexRDD[Msg]` que contiene el mensaje agregado (de tipo `Msg`) destinado a cada vértice. Los vértices que no recibieron un mensaje no se incluyen en el `VertexRDD`VertexRDD devuelto .

Además, `aggregateMessages`se utiliza un valor opcional `tripletsFields`que indica a qué datos se accede en [nombre `EdgeContext` del atributo de vértice de origen] (es decir, el atributo de vértice de origen, pero no el de destino). Las opciones posibles para [nombre del atributo de vértice de origen] `tripletsFields`se definen en [nombre del atributo de vértice de destino `TripletFields`] y el valor predeterminado es [nombre del atributo de vértice de destino `TripletFields.All`], lo que indica que la función definida por el usuario `sendMsg`puede acceder a cualquiera de los campos de [nombre del atributo de origen] `EdgeContext`. El `tripletFields`argumento se puede usar para notificar a GraphX que solo `EdgeContext`se necesitará una parte de [nombre del atributo de origen], lo que permite a GraphX seleccionar una estrategia de unión optimizada. Por ejemplo, si calculamos la edad promedio de los seguidores de cada usuario, solo requeriríamos el campo de origen, por lo que usaríamos [nombre del atributo de origen] `TripletFields.Src`para indicar que solo se requiere dicho campo.

En versiones anteriores de GraphX, utilizamos la inspección de código de bytes para inferir; `TripletFields`sin embargo, descubrimos que dicha inspección era ligeramente poco confiable y, en su lugar, optamos por un control de usuario más explícito.

En el siguiente ejemplo utilizamos el `aggregateMessages`operador para calcular la edad promedio de los seguidores más antiguos de cada usuario.

```
import org.apache.spark.graphx.{Graph, VertexRDD}
import org.apache.spark.graphx.util.GraphGenerators

// Create a graph with "age" as the vertex property.
// Here we use a random graph for simplicity.
val graph: Graph[Double, Int] =
  GraphGenerators.logNormalGraph(sc, numVertices = 100).mapVertices( (id, _) =>
id.toDouble )
// Compute the number of older followers and their total age
val olderFollowers: VertexRDD[(Int, Double)] = graph.aggregateMessages[(Int,
Double)](
  triplet => { // Map Function
    if (triplet.srcAttr > triplet.dstAttr) {
      // Send message to destination vertex containing counter and age
      triplet.sendToDst((1, triplet.srcAttr))
```

```
    }
  },
  // Add counter and age
  (a, b) => (a._1 + b._1, a._2 + b._2) // Reduce Function
)
// Divide total age by number of older followers to get average age of older followers
val avgAgeOfOlderFollowers: VertexRDD[Double] =
  olderFollowers.mapValues( (id, value) =>
    value match { case (count, totalAge) => totalAge / count } )
// Display the results
avgAgeOfOlderFollowers.collect.foreach(println(_))
```

Encuentre el código de ejemplo completo en "examples/src/main/scala/org/apache/spark/examples/graphx/AggregateMessagesExample.scala" en el repositorio de Spark.

La `aggregateMessages`operación funciona de manera óptima cuando los mensajes (y las sumas de mensajes) tienen un tamaño constante (por ejemplo, flotantes y sumas en lugar de listas y concatenación).

Guía de transición de tripletes de Map Reduce (Legacy)

En versiones anteriores de GraphX, la agregación de vecindarios se lograba mediante el `mapReduceTriplets`operador:

```
class Graph[VD, ED] {
  def mapReduceTriplets[Msg](
      map: EdgeTriplet[VD, ED] => Iterator[(VertexId, Msg)],
      reduce: (Msg, Msg) => Msg)
    : VertexRDD[Msg]
}
```

El `mapReduceTriplets`operador utiliza una función de mapa definida por el usuario que se aplica a cada triplete y puede generar *mensajes* que se agregan mediante dicha `reduce`función. Sin embargo, observamos que el uso del iterador devuelto resultaba costoso y limitaba la posibilidad de aplicar optimizaciones adicionales (por ejemplo, la renumeración de vértices locales). En [`aggregateMessages`Introducción], introdujimos EdgeContext, que expone los campos del triplete y también funciona para enviar mensajes explícitamente a los vértices de origen y destino. Además, eliminamos la inspección de bytecode y, en su lugar, requeríamos que el usuario indicara qué campos del triplete son realmente necesarios.

El siguiente bloque de código utiliza `mapReduceTriplets`:

```
val graph: Graph[Int, Float] = ...
def msgFun(triplet: Triplet[Int, Float]): Iterator[(Int, String)] = {
  Iterator((triplet.dstId, "Hi"))
```

```
}
def reduceFun(a: String, b: String): String = a + " " + b
val result = graph.mapReduceTriplets[String](msgFun, reduceFun)
```

se puede reescribir usando `aggregateMessages`como:

```
val graph: Graph[Int, Float] = ...
def msgFun(triplet: EdgeContext[Int, Float, String]) {
  triplet.sendToDst("Hi")
}
def reduceFun(a: String, b: String): String = a + " " + b
val result = graph.aggregateMessages[String](msgFun, reduceFun)
```

Información sobre el Grado en Informática

Una tarea común de agregación es calcular el grado de cada vértice: el número de aristas adyacentes a cada vértice. En el contexto de grafos dirigidos, suele ser necesario conocer el grado de entrada, el grado de salida y el grado total de cada vértice. La `GraphOps`clase contiene una colección de operadores para calcular los grados de cada vértice. Por ejemplo, a continuación, calculamos el máximo de grados de entrada, de salida y el total:

```
// Define a reduce operation to compute the highest degree vertex
def max(a: (VertexId, Int), b: (VertexId, Int)): (VertexId, Int) = {
  if (a._2 > b._2) a else b
}
// Compute the max degrees
val maxInDegree: (VertexId, Int)  = graph.inDegrees.reduce(max)
val maxOutDegree: (VertexId, Int) = graph.outDegrees.reduce(max)
val maxDegrees: (VertexId, Int)   = graph.degrees.reduce(max)
```

Recopilando vecinos

En algunos casos, puede ser más fácil expresar el cálculo recopilando los vértices vecinos y sus atributos en cada vértice. Esto se puede lograr fácilmente utilizando los operadores `collectNeighborIds`y `.collectNeighbors`

```
class GraphOps[VD, ED] {
  def collectNeighborIds(edgeDirection: EdgeDirection):
VertexRDD[Array[VertexId]]
  def collectNeighbors(edgeDirection: EdgeDirection): VertexRDD[
Array[(VertexId, VD)] ]
}
```

Estos operadores pueden ser bastante costosos, ya que duplican información y requieren mucha comunicación. Si es posible, intente expresar el mismo cálculo utilizando el `aggregateMessages` operador directamente.

Almacenamiento en caché y descatalogado

En Spark, los RDD no se almacenan en memoria de forma predeterminada. Para evitar recálculos, deben almacenarse en caché explícitamente al usarse varias veces (consulte la Guía de programación de Spark). Los gráficos en GraphX se comportan de la misma manera. **Al usar un gráfico varias veces, asegúrese de llamarlo `Graph.cache()` primero.**

En los cálculos iterativos, *la descaché* también puede ser necesaria para un rendimiento óptimo. De forma predeterminada, los RDD y grafos en caché permanecerán en memoria hasta que la presión de memoria los obligue a ser desalojados en orden LRU. En el cálculo iterativo, los resultados intermedios de iteraciones anteriores llenarán la caché. Aunque eventualmente se desalojarán, los datos innecesarios almacenados en memoria ralentizarán la recolección de elementos no utilizados. Sería más eficiente descaché los resultados intermedios en cuanto dejen de ser necesarios. Esto implica materializar (almacenar en caché y forzar) un grafo o RDD en cada iteración, descaché todos los demás conjuntos de datos y usar solo el conjunto de datos materializado en iteraciones futuras. Sin embargo, dado que los grafos se componen de múltiples RDD, puede ser difícil despersistirlos correctamente. **Para el cálculo iterativo, recomendamos usar la API de Pregel, que despersistirá correctamente los resultados intermedios.**

API de pregel

Los grafos son estructuras de datos inherentemente recursivas, ya que las propiedades de los vértices dependen de las propiedades de sus vecinos, que a su vez dependen de las propiedades de *estos* . En consecuencia, muchos algoritmos de grafos importantes recalculan iterativamente las propiedades de cada vértice hasta alcanzar una condición de punto fijo. Se han propuesto diversas abstracciones de grafos paralelos para expresar estos algoritmos iterativos. GraphX expone una variante de la API Pregel.

A alto nivel, el operador Pregel en GraphX es una abstracción de mensajería paralela síncrona masiva, *restringida a la topología del grafo* . El operador Pregel se ejecuta en una serie de superpasos en los que los vértices reciben la *suma* de sus mensajes entrantes del superpaso anterior, calculan un nuevo valor para la propiedad del vértice y, a continuación, envían mensajes a los vértices vecinos en el siguiente superpaso. A diferencia de Pregel, los mensajes se calculan en paralelo en función del triplete de aristas, y el cálculo del mensaje tiene acceso a los atributos del vértice de origen y destino. Los vértices que no reciben un mensaje se omiten en un superpaso. El operador Pregel finaliza la iteración y devuelve el grafo final cuando no quedan mensajes.

Tenga en cuenta que, a diferencia de las implementaciones estándar de Pregel, los vértices en GraphX solo pueden enviar mensajes a los vértices vecinos, y la construcción de mensajes se realiza en paralelo mediante una función de mensajería definida por el usuario. Estas restricciones permiten una optimización adicional en GraphX.

Lo siguiente es la firma de tipo del operador Pregel , así como un *bosquejo* de su implementación (nota: para evitar stackOverflowError debido a largas cadenas de linaje,

pregel admite periódicamente el gráfico de puntos de control y los mensajes configurando "spark.graphx.pregel.checkpointInterval" en un número positivo, digamos 10. Y también configure el directorio del punto de control usando SparkContext.setCheckpointDir(directory: String)):

```
class GraphOps[VD, ED] {
  def pregel[A]
      (initialMsg: A,
       maxIter: Int = Int.MaxValue,
       activeDir: EdgeDirection = EdgeDirection.Out)
      (vprog: (VertexId, VD, A) => VD,
       sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
       mergeMsg: (A, A) => A)
    : Graph[VD, ED] = {
    // Receive the initial message at each vertex
    var g = mapVertices( (vid, vdata) => vprog(vid, vdata, initialMsg) ).cache()

    // compute the messages
    var messages = GraphXUtils.mapReduceTriplets(g, sendMsg, mergeMsg)
    var activeMessages = messages.count()
    // Loop until no messages remain or maxIterations is achieved
    var i = 0
    while (activeMessages > 0 && i < maxIterations) {
      // Receive the messages and update the vertices.
      g = g.joinVertices(messages)(vprog).cache()
      val oldMessages = messages
      // Send new messages, skipping edges where neither side received a
message. We must cache
      // messages so it can be materialized on the next line, allowing us to
uncache the previous
      // iteration.
      messages = GraphXUtils.mapReduceTriplets(
        g, sendMsg, mergeMsg, Some((oldMessages, activeDirection))).cache()
      activeMessages = messages.count()
      i += 1
    }
    g
  }
}
```

Observe que Pregel utiliza dos listas de argumentos (es decir, `graph.pregel(list1)(list2)`). La primera lista contiene parámetros de

configuración, como el mensaje inicial, el número máximo de iteraciones y la dirección de los bordes en los que se enviarán los mensajes (por defecto, a lo largo de los bordes). La segunda lista contiene las funciones definidas por el usuario para recibir mensajes (el programa vertex `vprog`), calcular mensajes (`sendMsg`) y combinarlos `mergeMsg`.

Podemos utilizar el operador Pregel para expresar cálculos como la ruta más corta de una sola fuente en el siguiente ejemplo.

```
import org.apache.spark.graphx.{Graph, VertexId}
import org.apache.spark.graphx.util.GraphGenerators


// A graph with edge attributes containing distances
val graph: Graph[Long, Double] =
  GraphGenerators.logNormalGraph(sc, numVertices = 100).mapEdges(e =>
e.attr.toDouble)
val sourceId: VertexId = 42 // The ultimate source
// Initialize the graph such that all vertices except the root have distance
 infinity.
val initialGraph = graph.mapVertices((id, _) =>
    if (id == sourceId) 0.0 else Double.PositiveInfinity)
val sssp = initialGraph.pregel(Double.PositiveInfinity)(
  (id, dist, newDist) => math.min(dist, newDist), // Vertex Program
  triplet => {  // Send Message
    if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
      Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
    } else {
      Iterator.empty
    }
  },
  (a, b) => math.min(a, b) // Merge Message
)
println(sssp.vertices.collect.mkString("\n"))
```

Encuentre el código de ejemplo completo en "examples/src/main/scala/org/apache/spark/examples/graphx/SSSPExample.scala" en el repositorio de Spark.

Constructores de gráficos

GraphX ofrece varias maneras de construir un grafo a partir de una colección de vértices y aristas en un RDD o en disco. Ninguno de los constructores de grafos reparticiona las aristas del grafo por defecto; en su lugar, las aristas se conservan en sus particiones predeterminadas (como sus bloques originales en HDFS). `Graph.groupEdges`requiere que el grafo se reparticione porque asume que las aristas idénticas se colocarán en la misma partición, por lo que debe llamar `Graph.partitionBy`antes de llamar a `groupEdges`.

```
object GraphLoader {
```

```
  def edgeListFile(
      sc: SparkContext,
      path: String,
      canonicalOrientation: Boolean = false,
      minEdgePartitions: Int = 1)
    : Graph[Int, Int]
}
```

`GraphLoader.edgeListFile`Proporciona una forma de cargar un grafo desde una lista de aristas en disco. Analiza una lista de adyacencia de pares (ID de vértice de origen, ID de vértice de destino) con la siguiente forma, omitiendo las líneas de comentario que empiezan por #:

```
# This is a comment
2 1
4 1
1 2
```

Crea una `Graph`a partir de las aristas especificadas, creando automáticamente cualquier vértice mencionado por las aristas. Todos los atributos de vértice y arista tienen el valor predeterminado 1. El `canonicalOrientation`argumento permite reorientar las aristas en la dirección positiva (`srcId < dstId`), requerida por el algoritmo de componentes conectados . El `minEdgePartitions`argumento especifica el número mínimo de particiones de arista que se generarán; puede haber más particiones de arista de las especificadas si, por ejemplo, el archivo HDFS tiene más bloques.

```
object Graph {
  def apply[VD, ED](
      vertices: RDD[(VertexId, VD)],
      edges: RDD[Edge[ED]],
      defaultVertexAttr: VD = null)
    : Graph[VD, ED]

  def fromEdges[VD, ED](
      edges: RDD[Edge[ED]],
      defaultValue: VD): Graph[VD, ED]

  def fromEdgeTuples[VD](
      rawEdges: RDD[(VertexId, VertexId)],
      defaultValue: VD,
      uniqueEdges: Option[PartitionStrategy] = None): Graph[VD, Int]

}
```

`Graph.apply`Permite crear un grafo a partir de RDD de vértices y aristas. Los vértices duplicados se seleccionan arbitrariamente y a los vértices que se encuentran en el RDD de arista, pero no en el de vértice, se les asigna el atributo predeterminado.

`Graph.fromEdges`permite crear un gráfico a partir de solo un RDD de aristas, creando automáticamente cualquier vértice mencionado por las aristas y asignándoles el valor predeterminado.

`Graph.fromEdgeTuples`Permite crear un grafo a partir de un RDD de tuplas de aristas, asignando a las aristas el valor 1 y creando automáticamente los vértices mencionados por las aristas, asignándoles el valor predeterminado. También admite la deduplicación de aristas; para ello, pase `Some``` `PartitionStrategy`como `uniqueEdges`parámetro (por ejemplo, `uniqueEdges = Some(PartitionStrategy.RandomVertexCut)`). Se requiere una estrategia de partición para colocar aristas idénticas en la misma partición y así poder deduplicarlas.

RDD de vértices y aristas

GraphX expone `RDD`vistas de los vértices y las aristas almacenadas en el grafo. Sin embargo, dado que GraphX mantiene los vértices y las aristas en estructuras de datos optimizadas y estas estructuras de datos proporcionan funcionalidad adicional, los vértices y las aristas se devuelven como `VertexRDD`VertexRDD y `EdgeRDD`EdgeRDD , respectivamente. En esta sección, revisamos algunas de las funciones útiles adicionales de estos tipos. Tenga en cuenta que esta es solo una lista incompleta; consulte la documentación de la API para obtener la lista oficial de operaciones.

VertexRDDs

La `VertexRDD[A]`extensión `RDD[(VertexId, A)]`y añade la restricción adicional de que cada una `VertexId`ocurre solo *una vez* . Además, `VertexRDD[A]`representa un *conjunto* de vértices, cada uno con un atributo de tipo `A`. Internamente, esto se logra almacenando los atributos de los vértices en una estructura de datos de mapa hash reutilizable. Por consiguiente, si dos `VertexRDD`s se derivan del mismo `VertexRDD`VertexRDD base (p. ej., mediante `filter`o `mapValues`), pueden unirse en tiempo constante sin evaluaciones hash. Para aprovechar esta estructura de datos indexada, `VertexRDD`VertexRDD ofrece la siguiente funcionalidad adicional:

```
class VertexRDD[VD] extends RDD[(VertexId, VD)] {
  // Filter the vertex set but preserves the internal index
  def filter(pred: Tuple2[VertexId, VD] => Boolean): VertexRDD[VD]
  // Transform the values without changing the ids (preserves the internal
index)
  def mapValues[VD2](map: VD => VD2): VertexRDD[VD2]
  def mapValues[VD2](map: (VertexId, VD) => VD2): VertexRDD[VD2]
  // Show only vertices unique to this set based on their VertexId's
  def minus(other: RDD[(VertexId, VD)])
  // Remove vertices from this set that appear in the other set
  def diff(other: VertexRDD[VD]): VertexRDD[VD]
```

```
  // Join operators that take advantage of the internal indexing to accelerate
joins (substantially)

  def leftJoin[VD2, VD3](other: RDD[(VertexId, VD2)])(f: (VertexId, VD,
Option[VD2]) => VD3): VertexRDD[VD3]

  def innerJoin[U, VD2](other: RDD[(VertexId, U)])(f: (VertexId, VD, U) => VD2):
VertexRDD[VD2]

  // Use the index on this RDD to accelerate a `reduceByKey` operation on the
input RDD.

  def aggregateUsingIndex[VD2](other: RDD[(VertexId, VD2)], reduceFunc: (VD2,
VD2) => VD2): VertexRDD[VD2]

}
```

Observe, por ejemplo, cómo el `filter`operador devuelve un `VertexRDD`VertexRDD . El filtro se implementa mediante un [nombre de la función - "a" `BitSet`], lo que reutiliza el índice y permite realizar uniones rápidas con otros [nombres de la función `VertexRDD`- "s"]. Asimismo, los `mapValues`operadores no permiten que la `map`función cambie [nombre de la función - "a"], lo que permite reutilizar `VertexId`las mismas estructuras de datos. Tanto [nombre de la función - "a"] como [nombres de la función - "a "] pueden identificar cuándo se unen dos [nombres de la función - "a"] derivados del mismo [nombres de la función - "a"] e implementar la unión mediante escaneo lineal en lugar de costosas búsquedas de puntos.`HashMap``leftJoin``innerJoin``VertexRDD``HashMap`

El `aggregateUsingIndex`operador es útil para la construcción eficiente de un nuevo `VertexRDD`VertexRDD a partir de un `RDD[(VertexId, A)]`. Conceptualmente, si he construido un `VertexRDD[B]`sobre un conjunto de vértices, *que es un superconjunto* de los vértices en algún `RDD[(VertexId, A)]`, entonces puedo reutilizar el índice para agregar e indexar posteriormente el `RDD[(VertexId, A)]`. Por ejemplo:

```
val setA: VertexRDD[Int] = VertexRDD(sc.parallelize(0L until 100L).map(id =>
(id, 1)))

val rddB: RDD[(VertexId, Double)] = sc.parallelize(0L until 100L).flatMap(id =>
List((id, 1.0), (id, 2.0)))

// There should be 200 entries in rddB

rddB.count

val setB: VertexRDD[Double] = setA.aggregateUsingIndex(rddB, _ + _)

// There should be 100 entries in setB

setB.count

// Joining A and B should now be fast!

val setC: VertexRDD[Double] = setA.innerJoin(setB)((id, a, b) => a + b)
```

EdgeRDD

El `EdgeRDD[ED]`, que extiende, `RDD[Edge[ED]]`organiza las aristas en bloques particionados mediante una de las diversas estrategias de partición definidas en `PartitionStrategy`. Dentro de cada partición, los atributos de arista y la estructura de adyacencia se almacenan por separado, lo que permite una máxima reutilización al cambiar los valores de los atributos.

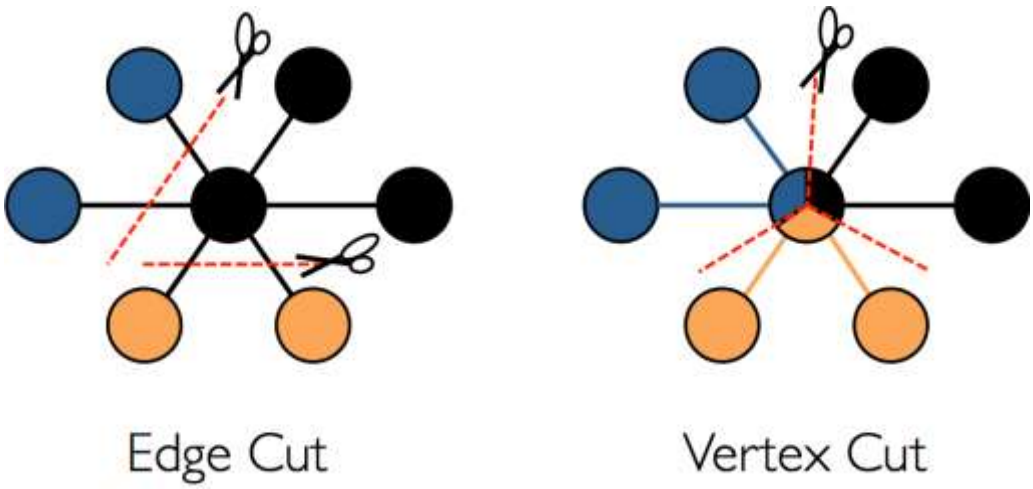
Las tres funciones adicionales expuestas por `EdgeRDD`EdgeRDD son:

```
// Transform the edge attributes while preserving the structure
def mapValues[ED2](f: Edge[ED] => ED2): EdgeRDD[ED2]
// Reverse the edges reusing both attributes and structure
def reverse: EdgeRDD[ED]
// Join two `EdgeRDD`s partitioned using the same partitioning strategy.
def innerJoin[ED2, ED3](other: EdgeRDD[ED2])(f: (VertexId, VertexId, ED, ED2) =>
ED3): EdgeRDD[ED3]
```

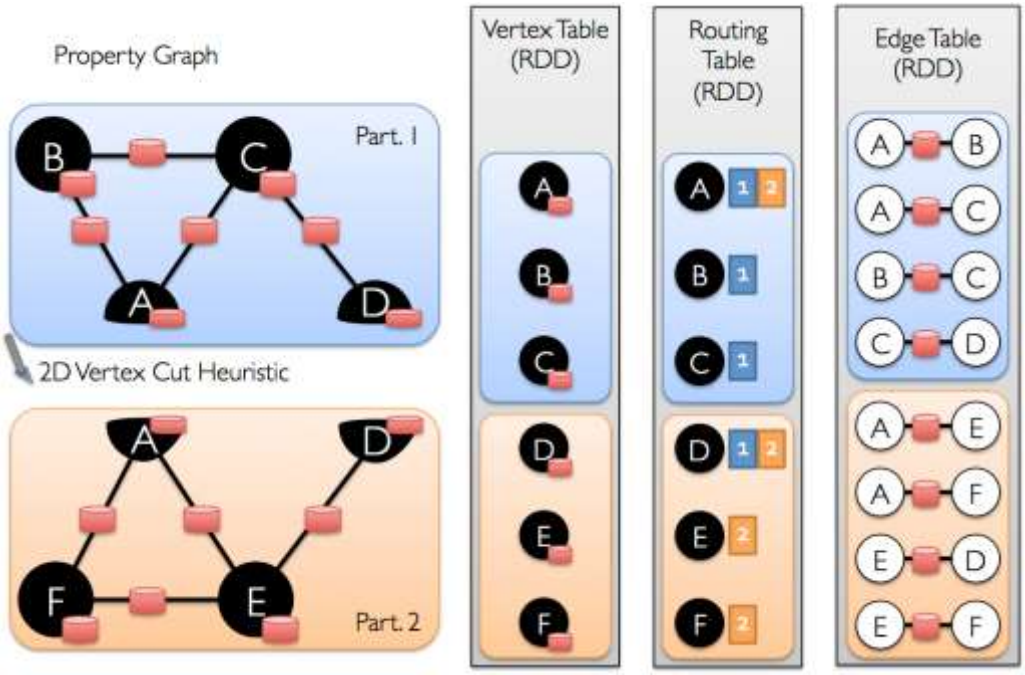
En la mayoría de las aplicaciones hemos descubierto que las operaciones en `EdgeRDD`EdgeRDD se realizan a través de operadores gráficos o dependen de operaciones definidas en la `RDD`clase base.

Representación optimizada

Si bien una descripción detallada de las optimizaciones utilizadas en la representación GraphX de grafos distribuidos queda fuera del alcance de esta guía, un conocimiento profundo puede facilitar el diseño de algoritmos escalables, así como el uso óptimo de la API. GraphX adopta un enfoque de corte de vértices para la partición de grafos distribuidos:



En lugar de dividir los grafos por aristas, GraphX particiona el grafo por vértices, lo que reduce la sobrecarga de comunicación y almacenamiento. Lógicamente, esto equivale a asignar aristas a las máquinas y permitir que los vértices abarquen varias máquinas. El método exacto de asignación de aristas depende de las `PartitionStrategy`distintas heurísticas, y existen varias ventajas y desventajas para cada una. Los usuarios pueden elegir entre diferentes estrategias reparticionando el grafo con el `Graph.partitionBy`operador. La estrategia de particionamiento predeterminada es utilizar la partición inicial de las aristas, tal como se proporciona en la construcción del grafo. Sin embargo, los usuarios pueden cambiar fácilmente a la partición 2D u otras heurísticas incluidas en GraphX.



Once the edges have been partitioned the key challenge to efficient graph-parallel computation is efficiently joining vertex attributes with the edges. Because real-world graphs typically have more edges than vertices, we move vertex attributes to the edges. Because not all partitions will contain edges adjacent to all vertices we internally maintain a routing table which identifies where to broadcast vertices when implementing the join required for operations like `triplets` and `aggregateMessages`.

Graph Algorithms

GraphX includes a set of graph algorithms to simplify analytics tasks. The algorithms are contained in the `org.apache.spark.graphx.lib` package and can be accessed directly as methods on `Graph` via `GraphOps`. This section describes the algorithms and how they are used.

PageRank

PageRank measures the importance of each vertex in a graph, assuming an edge from *u* to *v* represents an endorsement of *v*'s importance by *u*. For example, if a Twitter user is followed by many others, the user will be ranked highly.

GraphX comes with static and dynamic implementations of PageRank as methods on the `PageRank` object. Static PageRank runs for a fixed number of iterations, while dynamic PageRank runs until the ranks converge (i.e., stop changing by more than a specified tolerance). `GraphOps` allows calling these algorithms directly as methods on `Graph`.

GraphX also includes an example social network dataset that we can run PageRank on. A set of users is given in `data/graphx/users.txt`, and a set of relationships between users is given in `data/graphx/followers.txt`. We compute the PageRank of each user as follows:

```
import org.apache.spark.graphx.GraphLoader
```

```scala
// Load the edges as a graph
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt")
// Run PageRank
val ranks = graph.pageRank(0.0001).vertices
// Join the ranks with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val ranksByUsername = users.join(ranks).map {
  case (id, (username, rank)) => (username, rank)
}
// Print the result
println(ranksByUsername.collect().mkString("\n"))
```

Find full example code at "examples/src/main/scala/org/apache/spark/examples/graphx/PageRankExample.scala" in the Spark repo.

Connected Components

The connected components algorithm labels each connected component of the graph with the ID of its lowest-numbered vertex. For example, in a social network, connected components can approximate clusters. GraphX contains an implementation of the algorithm in the `ConnectedComponents` object, and we compute the connected components of the example social network dataset from the PageRank section as follows:

```scala
import org.apache.spark.graphx.GraphLoader

// Load the graph as in the PageRank example
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt")
// Find the connected components
val cc = graph.connectedComponents().vertices
// Join the connected components with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val ccByUsername = users.join(cc).map {
  case (id, (username, cc)) => (username, cc)
}
// Print the result
```

```
println(ccByUsername.collect().mkString("\n"))
```

Find full example code at "examples/src/main/scala/org/apache/spark/examples/graphx/ConnectedComponentsExample.scala" in the Spark repo.

Triangle Counting

Un vértice forma parte de un triángulo cuando tiene dos vértices adyacentes con una arista entre ellos. GraphX implementa un algoritmo de conteo de triángulos en el `TriangleCount`objeto que determina el número de triángulos que pasan por cada vértice, lo que proporciona una medida de agrupamiento. Calculamos el conteo de triángulos del conjunto de datos de redes sociales a partir de la sección PageRank . *Tenga en cuenta que `TriangleCount`requiere que las aristas estén en orientación canónica (`srcId < dstId`) y que el grafo se divida mediante `Graph.partitionBy`.*

```
import org.apache.spark.graphx.{GraphLoader, PartitionStrategy}

// Load the edges in canonical order and partition the graph for triangle count
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt", true)
  .partitionBy(PartitionStrategy.RandomVertexCut)
// Find the triangle count for each vertex
val triCounts = graph.triangleCount().vertices
// Join the triangle counts with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val triCountByUsername = users.join(triCounts).map { case (id, (username, tc))
=>
  (username, tc)
}
// Print the result
println(triCountByUsername.collect().mkString("\n"))
```

Encuentre el código de ejemplo completo en "examples/src/main/scala/org/apache/spark/examples/graphx/TriangleCountingExample.scala" en el repositorio de Spark.

Ejemplos

Supongamos que quiero crear un gráfico a partir de archivos de texto, restringirlo a relaciones y usuarios importantes, ejecutar PageRank en el subgráfico y, finalmente, devolver los atributos asociados con los usuarios principales. Puedo hacer todo esto en tan solo unas líneas con GraphX:

```
import org.apache.spark.graphx.GraphLoader
```

```scala
// Load my user data and parse into tuples of user id and attribute list
val users = (sc.textFile("data/graphx/users.txt")
  .map(line => line.split(",")).map( parts => (parts.head.toLong, parts.tail) ))

// Parse the edge data which is already in userId -> userId format
val followerGraph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt")

// Attach the user attributes
val graph = followerGraph.outerJoinVertices(users) {
  case (uid, deg, Some(attrList)) => attrList
  // Some users may not have attributes so we set them as empty
  case (uid, deg, None) => Array.empty[String]
}

// Restrict the graph to users with usernames and names
val subgraph = graph.subgraph(vpred = (vid, attr) => attr.size == 2)

// Compute the PageRank
val pagerankGraph = subgraph.pageRank(0.001)

// Get the attributes of the top pagerank users
val userInfoWithPageRank = subgraph.outerJoinVertices(pagerankGraph.vertices) {
  case (uid, attrList, Some(pr)) => (pr, attrList.toList)
  case (uid, attrList, None) => (0.0, attrList.toList)
}

println(userInfoWithPageRank.vertices.top(5)(Ordering.by(_._2._1)).mkString("\n"
))
```

Encuentre el código de ejemplo completo en "examples/src/main/scala/org/apache/spark/examples/graphx/ComprehensiveExample.scala" en el repositorio de Spark.