Spark pregel

如果对spark graphx有较深的理解那个理解pregel接口就相当简单了，因为就10几行代码。

# Spark pregel接口

首先需要理解一下pregel是啥？能干嘛？

Pregel是googel的大型图计算模型，主要用在图计算的迭代算法中。

spark pregel 的接口：

Pregel[VD, ED, A](

graph: Graph[VD, ED],

initialMsg: A,

maxIterations: Int = Int.MaxValue,

activeDirection: EdgeDirection = EdgeDirection.Either

)

(

vprog: (VertexId, VD, A) ⇒ VD,

sendMsg: (EdgeTriplet[VD, ED]) ⇒ Iterator[(VertexId, A)],

mergeMsg: (A, A) ⇒ A

): Graph[VD, ED]

参数：

**VD** the vertex data type

**ED** the edge data type

**A** the Pregel message type

**graph** the input graph.

**initialMsg** the message each vertex will receive at the first iteration

**maxIterations** the maximum number of iterations to run for

**activeDirection** the direction of edges incident to a vertex that received a message in the previous round on which to run sendMsg. For example, if this isEdgeDirection.Out, only out-edges of vertices that received a message in the previous round will run. The default is EdgeDirection.Either, which will run sendMsg on edges where either side received a message in the previous round. If this is EdgeDirection.Both, sendMsgwill only run on edges where \*both\* vertices received a message.

**vprog** the user-defined vertex program which runs on each vertex and receives the inbound message and computes a new vertex value. On the first iteration the vertex program is invoked on all vertices and is passed the default message. On subsequent iterations the vertex program is only invoked on those vertices that receive messages.

**sendMsg** a user supplied function that is applied to out edges of vertices that received messages in the current iteration

**mergeMsg** a user supplied function that takes two incoming messages of type A and merges them into a single message of type A. *This function must be commutative and associative and ideally the size of A should not increase.*

**returns** the resulting graph at the end of the computation

# Pregel的实现

def apply[VD: ClassTag, ED: ClassTag, A: ClassTag]

(graph: Graph[VD, ED],

initialMsg: A,

maxIterations: Int = Int.MaxValue,

activeDirection: EdgeDirection = EdgeDirection.Either)

(vprog: (VertexId, VD, A) => VD,

sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],

mergeMsg: (A, A) => A)

: Graph[VD, ED] =

{

require(maxIterations > 0, s"Maximum number of iterations must be greater than 0," +

s" but got ${maxIterations}")

var g = graph.mapVertices((vid, vdata) => vprog(vid, vdata, initialMsg)).cache()

// compute the messages

var messages = GraphXUtils.mapReduceTriplets(g, sendMsg, mergeMsg)

var activeMessages = messages.count()

// Loop

var prevG: Graph[VD, ED] = null

var i = 0

while (activeMessages > 0 && i < maxIterations) {

// Receive the messages and update the vertices.

prevG = g

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()

// The call to count() materializes `messages` and the vertices of `g`. This hides oldMessages

// (depended on by the vertices of g) and the vertices of prevG (depended on by oldMessages

// and the vertices of g).

activeMessages = messages.count()

logInfo("Pregel finished iteration " + i)

// Unpersist the RDDs hidden by newly-materialized RDDs

oldMessages.unpersist(blocking = false)

prevG.unpersistVertices(blocking = false)

prevG.edges.unpersist(blocking = false)

// count the iteration

i += 1

}

messages.unpersist(blocking = false)

g

} // end of apply

从Pregel的实现可以看见，pregel接口就封装了一个迭代算法，迭代次数由maxIterations决定。

Pregel接口接受一个图，根据initialMsg将其初始化成一个新图，其中顶点数据格式是(VertexId, VD, A)。

对图中每个顶点进行计算，使用sendMsg函数发送消息， mergeMsg函数合并消息，将此过程进行迭代就形成了pregel接口。

# Spark pregel pagerank

Pregel在pagerank算法上的应用，pagerank不懂看**<http://www.cnblogs.com/clover-siyecao/p/5726480.html>**

代码如下：

val pagerankGraph: Graph[Double, Double] = graph

// Associate the degree with each vertex

.outerJoinVertices(graph.outDegrees) {

(vid, vdata, deg) => deg.getOrElse(0)

}

// Set the weight on the edges based on the degree

.mapTriplets(e => 1.0 / e.srcAttr)

// Set the vertex attributes to the initial pagerank values

.mapVertices((id, attr) => 1.0)

def vertexProgram(id: VertexId, attr: Double, msgSum: Double): Double =

resetProb + (1.0 - resetProb) \* msgSum

def sendMessage(id: VertexId, edge: EdgeTriplet[Double, Double]): Iterator[(VertexId, Double)] =

Iterator((edge.dstId, edge.srcAttr \* edge.attr))

def messageCombiner(a: Double, b: Double): Double = a + b

val initialMessage = 0.0

// Execute Pregel for a fixed number of iterations.

Pregel(pagerankGraph, initialMessage, numIter)(

vertexProgram, sendMessage, messageCombiner)

**参考资料：**

<http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.graphx.Pregel$@apply[VD,ED,A](graph:org.apache.spark.graphx.Graph[VD,ED],initialMsg:A,maxIterations:Int,activeDirection:org.apache.spark.graphx.EdgeDirection)(vprog:(org.apache.spark.graphx.VertexId,VD,A)=>VD,sendMsg:org.apache.spark.graphx.EdgeTriplet[VD,ED]=>Iterator[(org.apache.spark.graphx.VertexId,A)],mergeMsg:(A,A)=>A)(implicitevidence$1:scala.reflect.ClassTag[VD],implicitevidence$2:scala.reflect.ClassTag[ED],implicitevidence$3:scala.reflect.ClassTag[A]):org.apache.spark.graphx.Graph[VD,ED]>

<http://spark.apache.org/docs/latest/graphx-programming-guide.html#pregel-api>

http://kowshik.github.io/JPregel/pregel\_paper.pdf

**参考代码：**

https://github.com/jie147/spark/blob/master/graphx/src/main/scala/org/apache/spark/graphx/Pregel.scala