**一、**

**1 介绍**

**1．1什么是PageRank**

PageRank是Google专有的算法，用于衡量特定网页相对于搜索引擎索引中的其他网页而言的重要程度。它由Larry Page 和 Sergey Brin在20世纪90年代后期发明。PageRank实现了将链接价值概念作为排名因素。PageRank将对页面的链接看成投票，指示了重要性。

**1．2 PageRank的应用场景**

搜索引擎难题，对搜索结果按重要性排序。

**对比：**

许多大公司就通过统计网页的访问量来进行网页排名。

**问题：**

一是因为只能够抽样统计，所以统计数据不一定准确，而且访问量的波动会比较大，想要得到准确的统计需要大量的时间和人力，还只能维持很短的有效时间；

二是访问量并不一定能体现网页的“重要程度”

**1．3 PageRank的逻辑和计算过程**

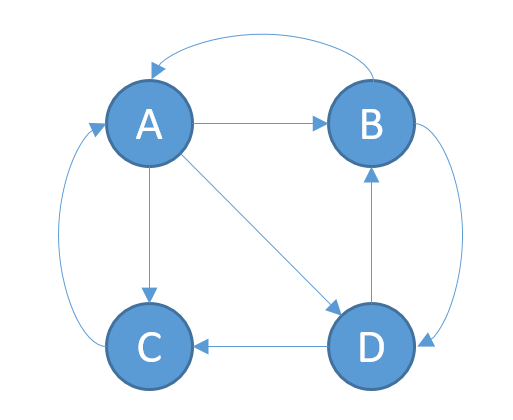
在互联网上，如果一个网页被很多其它网页所链接，说明它受到普遍的承认和依赖，那么它的排名就很高，也就更容易被访问到。

排名高 ＜＝＞ 访问概率大

怎么计算一个网页的访问概率？

1. 网页和网页之间的关系用图来表示
2. 网页A和网页B之间的连接关系表示任意一个用户从网页A跳转到网页B

**网页图**



**概率矩阵**

=

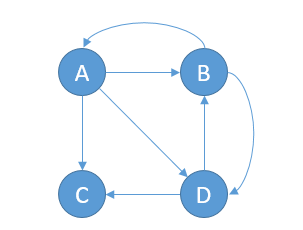
···

**结果**

收敛在

**问题：**

1. 终止点问题

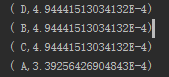




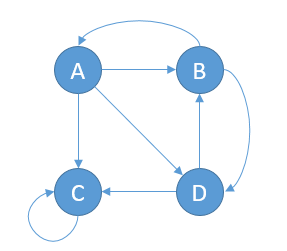
···

···

迭代20轮结果：



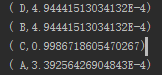
1. 陷阱问题





···

迭代20轮结果：



**改进方案**

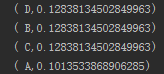
每个用户无论在任何页面，都有以一个较小概率跳转至其他页面。设为β=0.2，则有

1. 解决终止点问题

=

···

迭代20轮结果：

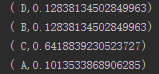


1. 解决陷阱问题

=

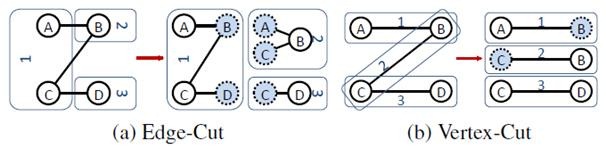
···

迭代20轮结果：



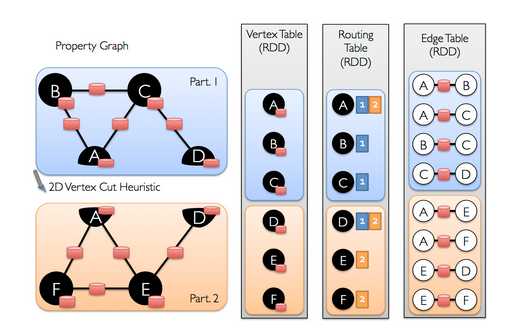
二、 **算法并行化**

**2．1 顶点分割 vs 边分割**



边分割（Edge-Cut）：每个顶点都存储一次，但有的边会被打断分到两台机器上。这样做的好处是节省存储空间；坏处是对图进行基于边的计算时，对于一条两个顶点被分到不同机器上的边来说，要跨机器通信传输数据，内网通信流量大。

点分割（Vertex-Cut）：每条边只存储一次，都只会出现在一台机器上。邻居多的点会被复制到多台机器上，增加了存储开销，同时会引发数据同步问题。好处是可以大幅减少内网通信量



1. EdgePartition1D

使用源顶点的VertexId分配边到分区，相同源顶点的边会被划分到一个分区。

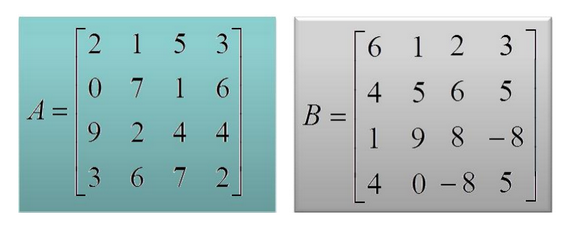
1. EdgePartition2D
2. RandomVertexCut

通过对源顶点和目标顶点的VertexId求取哈希值，然后除以分区数numParts取余数

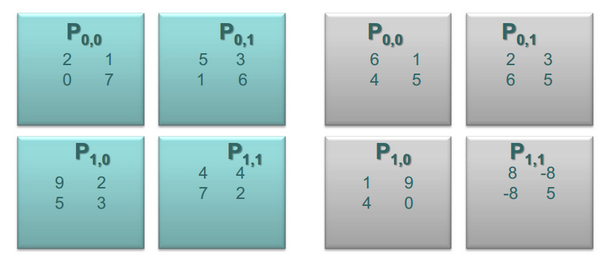
1. CanonicaRandomVertexCut

**2．2 矩阵并行计算**

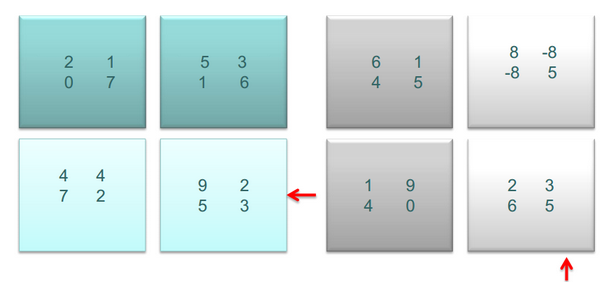
矩阵A\*B



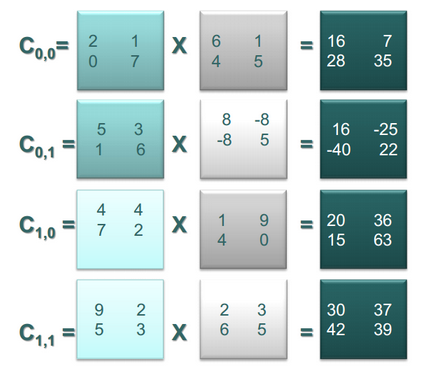
第一步：划分



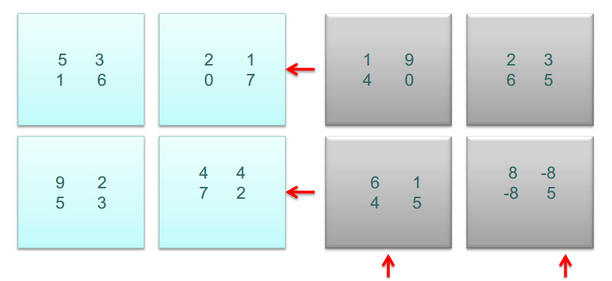
第二步：对齐



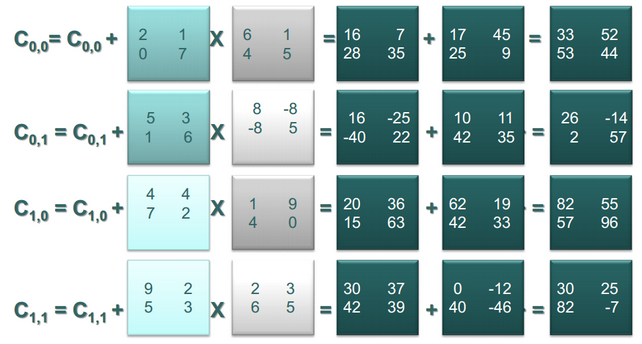
第三步：子矩阵相乘



第四步：对齐



第五步：计算结果



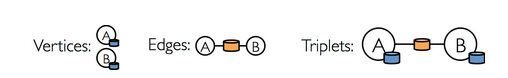
三、**Spark实现**

**3．1 图表示**

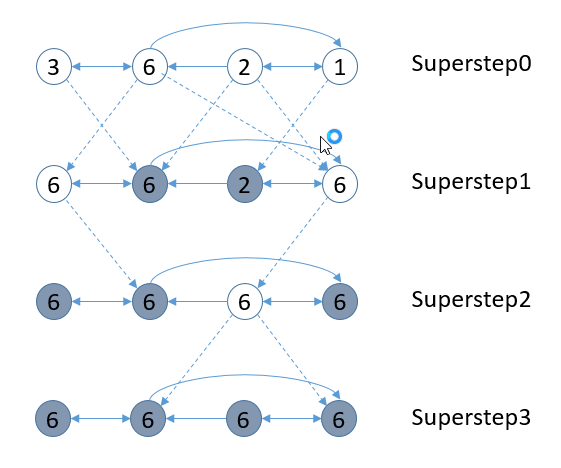
Edge : EdgeRDD[ED] RDD(Edge)

Vertice : VertexRDD[VD] RDD((vertexId, VD))

EdgeTriplet : RDD[EdgeTriplet[VD, ED]]



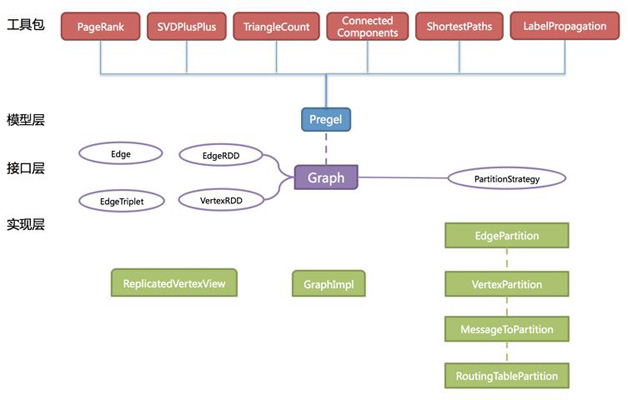
**3．2 迭代Pregel**



Pregel源码

**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  
}

**3．3 GraphX的代码结构**



**3．4 PageRank**

**1）简单实现**

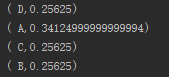
**val** links = sc.textFile("data/graphx/edge").map{ line =>  
 **val** fields = line.split("\\s+")  
 (fields(0), fields(1))  
}.distinct().groupByKey().persist();  
  
**var** ranks = sc.textFile("data/graphx/vertice").map{ line =>  
 **val** fields = line.split(",")  
 (fields(0), 1.0)  
}  
**val** N = ranks.count;  
ranks = ranks.mapValues(rank => rank/N)  
**val** delta = 0.0  
**val** iters = 10  
**for**(i <- 1 to iters) {  
 **val** contribs = links.join(ranks).flatMap{ **case** (src, (dests, rank)) =>  
 dests.map(dest => (dest, rank / dests.size))  
 }  
 ranks = contribs.reduceByKey((x, y) => x+y).mapValues(sum => delta/N + (1-delta)\*sum)  
}

**2）Spark实现**

**举例**

graph.staticPageRank(1)

**结果**



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | A | B | C | D |
| 1 | 0.34124999999999994 | 0.25625 | 0.25625 | 0.25625 |
| 2 | 0.47671874999999997 | 0.35559374999999993 | 0.35559374999999993 | 0.35559374999999993 |
| 3 | 0.6033820312499999 | 0.43619765624999995 | 0.43619765624999995 | 0.43619765624999995 |
| ··· |  |  |  |  |
| 50 | 1.2979104140828797 | 0.9003613286868322 | 0.9003613286868322 | 0.9003613286868322 |

**计算过程**

1. 初始化
2. 顶点attr (newPR, newPR-oldPR)
3. 构图初始化顶点attr (0.0, 0.0)
4. pagerank初始化消息：resetProb / (1 - resetProb)
5. 合并消息 a + b
6. 顶点处理函数newPR = oldPR + (1 - resetProb) \* msgSum

② 每轮

a. 发送消息 (newPR-oldPR) / outdeg

b. 合并消息 a + b

c. 顶点处理函数 newPR = oldPR + (1 - resetProb) \* msgSum

**上例具体计算过程**

第一轮

(0.0, 0.0) -> (0.15, 0.15)

A: 0.15 + (1 – 0.15) \* (0.15 / 2 + 0.15) = 0.34125 (0.34125, 0.19125)

B: 0.15 + (1 – 0.15) \* (0.15 / 3 + 0.15 / 2) = 0.25625 (0.25625, 0.10625)

C:

D:

第二轮

A: 0.34125 + (1 – 0.15) \* (0. 10625 / 2 + 0.10625) = 0.476718745

B: 0.25625 + (1 – 0.15) \* (0.19125 / 3 + 0.10625 / 2) = 0.35559375

C:

D:

···

**静态PageRank**

**def** runWithOptions[VD: ClassTag, ED: ClassTag](  
 graph: Graph[VD, ED], numIter: Int, resetProb: Double = 0.15,  
 srcId: Option[VertexId] = None): Graph[Double, Double] =  
{  
 *require*(numIter > 0, s"Number of iterations must be greater than 0," +  
 s" but got **$**{numIter}")  
 *require*(resetProb >= 0 && resetProb <= 1, s"Random reset probability must belong" +  
 s" to [0, 1], but got **$**{resetProb}")  
  
 **val** personalized = srcId isDefined  
 **val** src: VertexId = srcId.getOrElse(-1L)  
  
 // Initialize the PageRank graph with each edge attribute having  
 // weight 1/outDegree and each vertex with attribute resetProb.  
 // When running personalized pagerank, only the source vertex  
 // has an attribute resetProb. All others are set to 0.  
 **var** rankGraph: 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*, TripletFields.*Src* )  
 // Set the vertex attributes to the initial pagerank values  
 .mapVertices { (id, attr) =>  
 **if** (!(id != src && personalized)) resetProb **else** 0.0  
 }  
  
 **def** delta(u: VertexId, v: VertexId): Double = { **if** (u == v) 1.0 **else** 0.0 }  
  
 **var** iteration = 0  
 **var** prevRankGraph: Graph[Double, Double] = **null  
 while** (iteration < numIter) {  
 rankGraph.cache()  
  
 // Compute the outgoing rank contributions of each vertex, perform local preaggregation, and  
 // do the final aggregation at the receiving vertices. Requires a shuffle for aggregation.  
 **val** rankUpdates = rankGraph.aggregateMessages[Double](  
 ctx => ctx.sendToDst(ctx.srcAttr \* ctx.attr), \_ + \_, TripletFields.*Src*)  
  
 // Apply the final rank updates to get the new ranks, using join to preserve ranks of vertices  
 // that didn't receive a message. Requires a shuffle for broadcasting updated ranks to the  
 // edge partitions.  
 prevRankGraph = rankGraph  
 **val** rPrb = **if** (personalized) {  
 (src: VertexId, id: VertexId) => resetProb \* delta(src, id)  
 } **else** {  
 (src: VertexId, id: VertexId) => resetProb  
 }  
  
 rankGraph = rankGraph.joinVertices(rankUpdates) {  
 (id, oldRank, msgSum) => rPrb(src, id) + (1.0 - resetProb) \* msgSum  
 }.cache()  
  
 rankGraph.edges.foreachPartition(x => {}) // also materializes rankGraph.vertices  
 logInfo(s"PageRank finished iteration **$**iteration.")  
 prevRankGraph.vertices.unpersist(**false**)  
 prevRankGraph.edges.unpersist(**false**)  
  
 iteration += 1  
 }  
  
 rankGraph  
}

**动态PageRank**

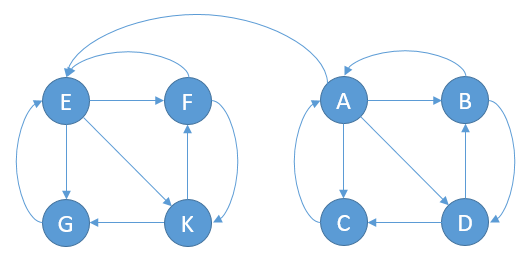
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | A | B | C | D |
| 0.1 | 0.47671874999999997 | 0.39397656249999996 | 0.39397656249999996 | 0.39397656249999996 |
| 0.05 | 0.9349622619168088 | 0.6765600615153653 | 0.6765600615153653 | 0.6765600615153653 |
| 0.01 | 1.2140943660722072 | 0.8486915257445278 | 0.8486915257445278 | 0.8486915257445278 |
| ··· |  |  |  |  |
| 0.0001 | 1.2974901577834612 | 0.9001189306331344 | 0.9001189306331344 | 0.9001189306331344 |

**def** runUntilConvergenceWithOptions[VD: ClassTag, ED: ClassTag](  
 graph: Graph[VD, ED], tol: Double, resetProb: Double = 0.15,  
 srcId: Option[VertexId] = None): Graph[Double, Double] =  
{  
 *require*(tol >= 0, s"Tolerance must be no less than 0, but got **$**{tol}")  
 *require*(resetProb >= 0 && resetProb <= 1, s"Random reset probability must belong" +  
 s" to [0, 1], but got **$**{resetProb}")  
  
 **val** personalized = srcId.isDefined  
 **val** src: VertexId = srcId.getOrElse(-1L)  
  
 // Initialize the pagerankGraph with each edge attribute  
 // having weight 1/outDegree and each vertex with attribute 1.0.  
 **val** pagerankGraph: Graph[(Double, 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 (initialPR, delta = 0)  
 .mapVertices { (id, attr) =>  
 **if** (id == src) (resetProb, Double.*NegativeInfinity*) **else** (0.0, 0.0)  
 }  
 .cache()  
  
 // Define the three functions needed to implement PageRank in the GraphX  
 // version of Pregel  
 **def** vertexProgram(id: VertexId, attr: (Double, Double), msgSum: Double): (Double, Double) = {  
 **val** (oldPR, lastDelta) = attr  
 **val** newPR = oldPR + (1.0 - resetProb) \* msgSum  
 (newPR, newPR - oldPR)  
 }  
  
 **def** personalizedVertexProgram(id: VertexId, attr: (Double, Double),  
 msgSum: Double): (Double, Double) = {  
 **val** (oldPR, lastDelta) = attr  
 **var** teleport = oldPR  
 **val** delta = **if** (src==id) 1.0 **else** 0.0  
 teleport = oldPR\*delta  
  
 **val** newPR = teleport + (1.0 - resetProb) \* msgSum  
 **val** newDelta = **if** (lastDelta == Double.*NegativeInfinity*) newPR **else** newPR - oldPR  
 (newPR, newDelta)  
 }  
  
 **def** sendMessage(edge: EdgeTriplet[(Double, Double), Double]) = {  
 **if** (edge.*srcAttr*.\_2 > tol) {  
 *Iterator*((edge.dstId, edge.*srcAttr*.\_2 \* edge.attr))  
 } **else** {  
 *Iterator*.*empty* }  
 }  
  
 **def** messageCombiner(a: Double, b: Double): Double = a + b  
  
 // The initial message received by all vertices in PageRank  
 **val** initialMessage = **if** (personalized) 0.0 **else** resetProb / (1.0 - resetProb)  
  
 // Execute a dynamic version of Pregel.  
 **val** vp = **if** (personalized) {  
 (id: VertexId, attr: (Double, Double), msgSum: Double) =>  
 personalizedVertexProgram(id, attr, msgSum)  
 } **else** {  
 (id: VertexId, attr: (Double, Double), msgSum: Double) =>  
 vertexProgram(id, attr, msgSum)  
 }  
  
 *Pregel*(pagerankGraph, initialMessage, activeDirection = EdgeDirection.*Out*)(  
 vp, sendMessage, messageCombiner)  
 .mapVertices((vid, attr) => attr.\_1)  
}

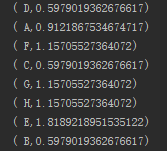
扩展性测试

|  |  |  |  |
| --- | --- | --- | --- |
| app | Vertice | Edge | Rum Time |
| pagerank | 100,000 | 233.3M | 44s |
|  | 100000 | 237.65M | 50s |
|  | 200000 | 506.93M | 1.2min |
|  | 200000 | 496.15M | 55s |
|  | 300000 | 767.42M | 1.1min |
|  | 300000 | 766.86M | 1.2min |
|  | 400000 | 1040.88M | 1.9min |
|  | 400000 | 1039.24M | 1.5min |
|  | 500000 | 1302.0M | 2.8min |
|  | 500000 | 1300.88M | 1.5min |

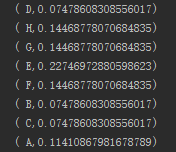
**3.两个分图，终止点和陷阱问题**



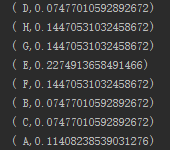
ResetProb = 0.15 停止迭代0.0001

排名：E,F,G,H,A,B,C,D

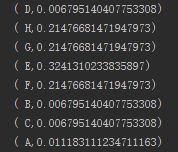
Delta = 0.15 迭代30轮

排名：E,F,G,H,A,B,C,D

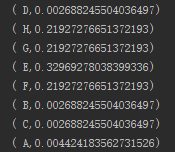
Delta = 0.15 迭代40轮



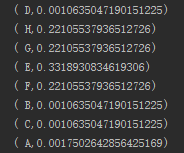
Delta = 0.0 迭代30轮



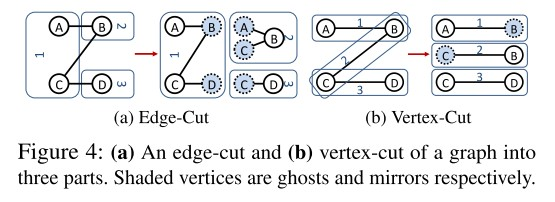
Delta = 0.0 迭代40轮



Delta = 0.0 迭代50轮



**5.图分割方法使用场景，为什么边分割节省空间？**



Edge-cut: 每个点vertex的数据都只存储在一台机器上，如例子中的点A，点A的完整数据只存放在机器1上，B的完整数据只存放在机器2上。

对于边AB而言，在机器1上，存放的是A的数据和B的索引地址（即B的数据在哪台机器上），在机器2上，存放的是B的数据和A的索引地址。

这样存储的好处是对于读取某个点的数据时，只要到一台机器上就可以了。缺点是当要读取一条边数据时，如果这条边的两个点在不同的机器上，那么就会引起跨机器的通信开销。当这个图是无标度网络时，这个开销将抵消节点数据只存储一份带来的好处(后面将详细说说)

Vertex-cut: graphx借鉴powerGraph，使用的是vertexcut(点分割)方式存储图。如上图b:这种存储方式特点是任何一条边只会出现在一台机器上，每个点有可能分布到不同的机器上，例如上图的点B就被分配到了2，3两条机器上。

当点被分割到不同机器上时，是相同的镜像，但是有一个点作为主点(master),其他的点作为虚点(ghost)，当点B的数据发生变化时,先更新点B的master的数据，然后将所有更新好的数据发送到B的ghost所在的所有机器，更新B的ghost。

这样做的好处是在边的存储上是没有冗余的，而且对于某个点与它的邻居的交互操作，只要满足交换律和结合律，比如求邻居权重的和，求点的所有边的条数这样的操作，可以在不同的机器上并行进行，只要把每个机器上的结果进行汇总就可以了，网络开销也比较小。代价是每个点可能要存储多份，更新点要有数据同步开销。

Routing Table就是用于点更新用的。

2.2:把图按照vertex-cut或edge-cut的存储的算法步骤

把图存储大体上有3中方法balanced p-way edge-cut,Balanced p-way Vertex-Cut,Greedy Vertex-Cuts

2.2.1:balanced p-way edge-cut

对于图G而言，按照balanced p-way edge cut的方式存储到p台机器上是这样做:先将vertex的ID平均hash到p台机器上，每个点都会唯一对应p台机器中的一个。然后将边一条一条加到集群上。遇到一条边的两个点在不同机器上的情况时，在两个机器上分别建立缺少的那个点的ghost

2.2.2:Balanced p-way Vertex-Cut

对于图G而言，按照balanced p-way edge cut的方式存储到p台机器上是这样做:先将edge的ID(比如使用两个端点的vertexId来合成出一个边的id)平均hash到p台机器上，每个条都会唯一对应p台机器中的一个。然后对于图中的每个点，如果它在多台机器上都有，选取一个点作为master，其他点作为ghost，将点的data分发到有这个点的所有机器上。

2.2.3:Greedy Vertex-Cuts

对于图G而言，按照Greedy Vertex-Cuts的方式存储到p台机器上是这样做:对于任意一条边e而言，设它的两个端点是a,b。点a被分配的机器集合是A(a),点a被分配的机器集合是A(a)。对于下面4种情况是这样操作的:

a:A(a)和A(b)的交集不为空，那么就把e分配到交集的机器上

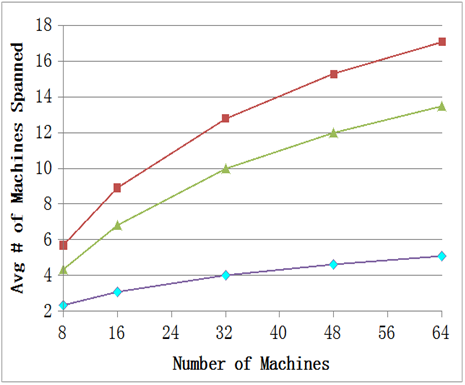
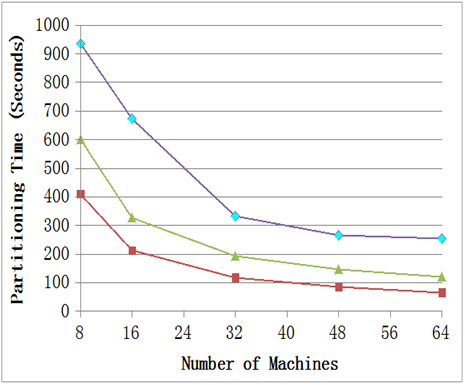
b:A(a),A(b)的交集为空，但是A(a),A(b)都不为空，那就把e分配到A(a)和A(b)的并集中已经分配到的边数最小的那台机器上。

c:如果a已经被分配过，b没有分配，那么就把e分配到有a的机器上，反之亦然

d:a,b都没有被分配过，那就把e分配到负载最小的机器上。

简单地说，这样做的目的就是让点尽可能分配得紧密一点，一个指定的点分配到包含这个点的边最多的那台机器上。

整体而言，Greedy Vertex-Cuts载入速度最慢，但是最终占用的空间最小，数据局部性效果最好，可以在随后的计算中带来速度上的收益,如下面的两张图



上图是卡内基梅隆大学将Twitter Graph: 41M vertices, 1.4B edges用上述方法载入到集群的速度和空间测试。红色是balanced p-way edge cut绿色是Balanced p-way Vertex-Cut紫色是Greedy Vertex-Cut,可见前期加载中的时间消耗会给后来的计算带来益处。这个原因大体上来说是因为Greedy Vertex-Cuts使得对于点的数据分配得更加集中，每个点平均会被分配到的机器数变小了，也就是点的平均拷贝数少了，占用的存储空间小了，减小了磁盘开销。同时因为点的数据集中，可以减少计算时用于数据同步的网络开销。

11.纠正报告中的join过程解释。

**7.Spark Graphx 为什么采用增量式 PageRank?**

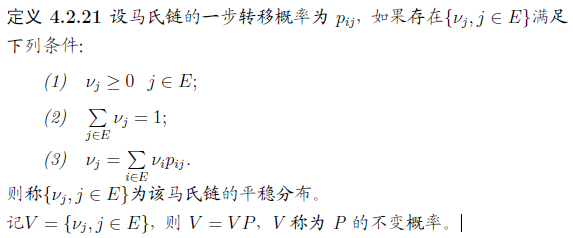
我认为和pregel计算模型有关，要发送消息，各个顶点停止时间不一致，这样使用原算法就很表达出来，如果仅是迭代次数的话，那没有问题。可能也与收敛速度和计算精度有关，因为按原算法算到最后，有些值太小，精度难以控制。

Note that this is not the "normalized" PageRank and as a consequence pages that have no inlinks will have a PageRank of alpha.

1. **pagerank为什么会收敛?**

概率矩阵称为转移概率矩阵

各网页的pagerank值构成为马氏链



对于上面的例子有

P = 

V = {A, B, C, D}

 得