快刀初试：Spark GraphX在淘宝的实践

发表于 2014-08-07 13:24 | 40051次阅读 | 来源 《程序员》 | 0 条评论 | 作者 黄明，吴炜

《程序员》杂志 2014年8月刊 Spark GraphX 淘宝 分布式图计算 大数据

摘要：由于Spark GraphX性能良好，又有丰富的功能和运算符，能在海量数据上自如运行复杂的图算法，淘宝尝试将它作为分布式图计算平台，进行各种算法尝试和生产应用。本文结合GraphX的原理和特点，分享其在淘宝的应用实践。

早在0.5版本，Spark就带了一个小型的Bagel模块，提供了类似Pregel的功能。当然，这个版本还非常原始，性能和功能都比较弱，属于实验型产品。到0.8版本时，鉴于业界对分布式图计算的需求日益见涨，Spark开始独立一个分支Graphx-Branch，作为独立的图计算模块，借鉴GraphLab，开始设计开发GraphX。在0.9版本中，这个模块被正式集成到主干，虽然是Alpha版本，但已可以试用，小面包圈Bagel告别舞台。1.0版本，GraphX正式投入生产使用。

值得注意的是，GraphX目前依然处于快速发展中，从0.8的分支到0.9和1.0，每个版本代码都有不少的改进和重构。根据观察，在没有改任何代码逻辑和运行环境，只是升级版本、切换接口和重新编译的情况下，每个版本有10%~20%的性能提升。虽然和GraphLab的性能还有一定差距，但凭借Spark整体上的一体化流水线处理，社区热烈的活跃度及快速改进速度，GraphX具有强大的竞争力。

分布式图计算

在正式介绍GraphX之前，先看看通用的分布式图计算框架。简单来说，分布式图计算框架的目的，是将对于巨型图的各种操作包装为简单的接口，让分布式存储、并行计算等复杂问题对上层透明，从而使复杂网络和图算法的工程师，更加聚焦在图相关的模型设计和使用上，而不用关心底层的分布式细节。为了实现该目的，需要解决两个通用问题：图存储模式和图计算模式。

图存储模式

巨型图的存储总体上有边分割和点分割两种存储方式。2013年，GraphLab2.0将其存储方式由边分割变为点分割，在性能上取得重大提升，目前基本上被业界广泛接受并使用。

边分割：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：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。

虽然两种方法互有利弊，但现在是点分割占上风，各种分布式图计算框架都将自己底层的存储形式变成了点分割。主要原因有以下两个。

- 磁盘价格下降，存储空间不再是问题，而内网的通信资源没有突破性进展，集群计算时内网带宽是宝贵的，时间比磁盘更珍贵。这点就类似于常见的空间换时间的策略。
- 在当前的应用场景中，绝大多数网络都是“无尺度网络”，遵循幂律分布，不同点的邻居数量相差非常悬殊。而边分割会使那些多邻居的点所相连的边大多数被分到不同的机器上，这样的数据分布会使得内网带宽更加捉襟见肘，于是边分割存储方式被渐渐抛弃了。

图计算模型

目前的图计算框架基本上都遵循BSP（Bulk Synchronous Parallell）计算模式。在BSP中，一次计算过程由一系列全局超步组成，每一个超步由并发计算、通信和栅栏同步三个步骤组成。同步完成，标

志着这个超步的完成及下一个超步的开始。BSP模式很简洁。基于BSP模式，目前有两种比较成熟的图计算模型。

- Pregel模型——像顶点一样思考

2010年，Google的新的三架马车Caffeine、Pregel、Dremel发布。随着Pregel一起，BSP模型广为人知。Pregel借鉴MapReduce的思想，提出了“像顶点一样思考”（Think Like A Vertex）的图计算模式，让用户无需考虑并行分布式计算的细节，只需要实现一个顶点更新函数，让框架在遍历顶点时进行调用即可。

常见的代码模板如下：

```
void Compute(MessageIterator* msgs) {
  // 遍历由顶点入边传入的消息列表
  for (; !msgs->Done(); msgs->Next())
    doSomething()
  // 生成新的顶点值
  *MutableVertexValue() = ...
  // 生成沿顶点出边发送的消息
  SendMessageToAllNeighbors(...);
}
```

这个模型虽然简洁，但很容易发现它的缺陷。对于邻居数很多的顶点，它需要处理的消息非常庞大，而且在这个模式下，它们是无法被并发处理的。所以对于符合幂律分布的自然图，这种计算模型下很容易发生假死或者崩溃。

- GAS模型——邻居更新模型

相比Pregel模型的消息通信范式，GraphLab的GAS模型更偏向共享内存风格。它允许用户的自定义函数访问当前顶点的整个邻域，可抽象成Gather、Apply和Scatter三个阶段，简称为GAS。相对应，用户需要实现三个独立的函数gather、apply和scatter。常见的代码模板如下所示：

```
// 从邻居点和边收集数据
Message gather(Vertex u, Edge uv, Vertex v)
{
  Message msg = ...
  return msg
}
// 汇总函数
Message sum(Message left, Message right) {
  return left+right
}
// 更新顶点Master
void apply(Vertex u, Message sum) {
  u.value = ...
}
// 更新邻边和邻居点
void scatter(Vertex u, Edge uv, Vertex v) {
  uv.value = ...
  if ((|u.delta|>ε) Active(v)
}
```

由于gather/scatter函数是以单条边为操作粒度，所以对于一个顶点的众多邻边，可以分别由相应的worker独立调用gather/scatter函数。这一设计主要是为了适应点分割的图存储模式，从而避免Pregel模型会遇到的问题。

GraphX的框架

在设计GraphX时，点分割和GAS都已成熟，并在设计和编码中针对它们进行了优化，在功能和性能之间寻找最佳的平衡点。如同Spark本身，每个子模块都有一个核心抽象。GraphX的核心抽象是Resilient Distributed Property Graph，一种点和边都带属性的有向多重图。它扩展了Spark RDD的抽象，有Table和Graph两种视图，而只需要一份物理存储。两种视图都有自己独有的操作符，从而获得了灵活操作和执行效率。

如同Spark，GraphX的代码非常简洁。GraphX的核心代码只有3千多行，而在此之上实现的Pregel模

型，只要短短的20多行。GraphX的代码结构整体如图1所示，其中大部分的实现，都是围绕Partition的优化进行的。这在某种程度上说明了点分割的存储和相应的计算优化，的确是图计算框架的重点和难点。

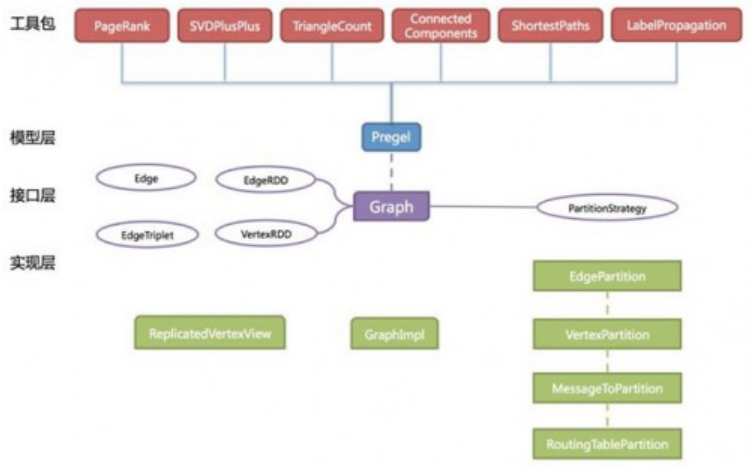

图1 GraphX的代码结构

GraphX的底层设计有以下几个关键点。

1. 对Graph视图的所有操作，最终都会转换成其关联的Table视图的RDD操作来完成。这样对一个图的计算，最终在逻辑上，等价于一系列RDD的转换过程。因此，Graph最终具备了RDD的3个关键特性：Immutable、Distributed和Fault-Tolerant。其中最关键的是Immutable（不变性）。逻辑上，所有图的转换和操作都产生了一个新图；物理上，GraphX会有一定程度的不变顶点和边的复用优化，对用户透明。
2. 两种视图底层共用的物理数据，由RDD[Vertex-Partition]和RDD[EdgePartition]这两个RDD组成。点和边实际都不是以表Collection[tuple]的形式存储的，而是由VertexPartition/EdgePartition在内部存储一个带索引结构的分片数据块，以加速不同视图下的遍历速度。不变的索引结构在RDD转换过程中是共用的，降低了计算和存储开销。
3. 图的分布式存储采用点分割模式，而且使用partitionBy方法，由用户指定不同的划分策略（PartitionStrategy）。划分策略会将边分配到各个EdgePartition，顶点Master分配到各个VertexPartition，EdgePartition也会缓存本地边关联点的Ghost副本。划分策略的不同会影响到所需要缓存的Ghost副本数量，以及每个EdgePartition分配的边的均衡程度，需要根据图的结构特征选取最佳策略。目前有EdgePartition2d、EdgePartition1d、RandomVertexCut和CanonicalRandomVertexCut这四种策略。在淘宝大部分场景下，EdgePartition2d效果最好。

GraphX的图运算符

如同Spark一样，GraphX的Graph类提供了丰富的图运算符，大致结构如图2所示。可以在官方GraphX Programming Guide中找到每个函数的详细说明，本文仅讲述几个需要注意的方法。

图2 GraphX的图运算符

图的cache

每个图是由3个RDD组成，所以会占用更多的内存。相应图的cache、unpersist和checkpoint，更需要注意使用技巧。出于最大限度复用边的理念，GraphX的默认接口只提供了unpersistVertices方法。

如果要释放边，调用g.edges.unpersist()方法才行，这给用户带来了一定的不便，但为GraphX的优化提供了便利和空间。参考GraphX的Pregel代码，对一个大图，目前最佳的实践是：

```
var g=...
var prevG: Graph[VD, ED] = null
while(...){
  prevG = g
  g = doSomething(g)
  g.cache()
  prevG.unpersistVertices(blocking=false)
  prevG.edges.unpersist(blocking=false)
}
```

大体之意是根据GraphX中Graph的不变性，对g做操作并赋回给g之后，g已不是原来的g了，而且会在下一轮迭代使用，所以必须cache。另外，必须先用prevG保留住对原来图的引用，并在新图产生后，快速将旧图彻底释放掉。否则，十几轮迭代后，会有内存泄漏问题，很快耗光作业缓存空间。

mrTriplets——邻边聚合

mrTriplets（mapReduceTriplets）是GraphX中最核心的一个接口。Pregel也基于它而来，所以对它的优化能很大程度上影响整个GraphX的性能。mrTriplets运算符的简化定义是：

```
def mapReduceTriplets[A](
  map: EdgeTriplet[VD, ED] =>
Iterator[(VertexID, A)],
  reduce: (A, A) => A)
  : VertexRDD[A]
```

它的计算过程为：map，应用于每一个Triplet上，生成一个或者多个消息，消息以Triplet关联的两个顶点中的任意一个或两个为目标顶点；reduce，应用于每一个Vertex上，将发送给每一个顶点的消息合并起来。

mrTriplets最后返回的是一个VertexRDD[A]，包含每一个顶点聚合之后的消息（类型为A），没有接收到消息的顶点不会包含在返回的VertexRDD中。

在最近的版本中，GraphX针对它进行了一些优化，对于Pregel以及所有上层算法工具包的性能都有重大影响。主要包括以下几点。

- **Caching for Iterative mrTriplets & Incremental Updates for Iterative mrTriplets**：在很多图分析算法中，不同点的收敛速度变化很大。在迭代后期，只有很少的点会有更新。因此，对于没有更新的点，下一次mrTriplets计算时EdgeRDD无需更新相应点值的本地缓存，大幅降低了通信开销。
- **Indexing Active Edges**：没有更新的顶点在下一轮迭代时不需要向邻居重新发送消息。因此，mrTriplets遍历边时，如果一条边的邻居点值在上一轮迭代时没有更新，则直接跳过，避免了大量无用的计算和通信。
- **Join Elimination**：Triplet是由一条边和其两个邻居点组成的三元组，操作Triplet的map函数常常只需访问其两个邻居点值中的一个。例如，在PageRank计算中，一个点值的更新只与其源顶点的值有关，而与其所指向的目的顶点的值无关。那么在mrTriplets计算中，就不需要VertexRDD和EdgeRDD的3-way join，而只需要2-way join。

所有这些优化使GraphX的性能逐渐逼近GraphLab。虽然还有一定差距，但一体化的流水线服务和丰富的编程接口，可以弥补性能的微小差距。

进化的**Pregel**模型

GraphX中的Pregel接口，并不严格遵循Pregel模型，它是一个参考GAS改进的Pregel模型。定义如下：

```
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]
```

这种基于mrTrilets方法的Pregel模型，与标准Pregel的最大区别是，它的第2段参数体接收的是3个函数参数，而不接收messageList。它不会在单个顶点上进行消息遍历，而是将顶点的多个Ghost副本收到的消息聚合后，发送给Master副本，再使用vprog函数来更新点值。消息的接收和发送都被自动并行化处理，无需担心超级节点的问题。

常见的代码模板如下所示：

```
// 更新顶点
vprog(vid: Long, vert: Vertex, msg: Double):
Vertex = {
  v.score = msg + (1 - ALPHA) * v.weight
}
// 发送消息
sendMsg(edgeTriplet: EdgeTriplet[…]):
Iterator[(Long, Double)]
  (destId, ALPHA * edgeTriplet.srcAttr.
score * edgeTriplet.attr.weight)
}
// 合并消息
mergeMsg(v1: Double, v2: Double): Double = {
  v1+v2
}
```

可以看到，GraphX设计这个模型的用意。它综合了Pregel和GAS两者的优点，即接口相对简单，又保证性能，可以应对点分割的图存储模式，胜任符合幂律分布的自然图的大型计算。另外，值得注意的是，官方的Pregel版本是最简单的一个版本。对于复杂的业务场景，根据这个版本扩展一个定制的Pregel是很常见的做法。

图算法工具包

GraphX也提供了一套图算法工具包，方便用户对图进行分析。目前最新版本已支持PageRank、数三角形、最大连通图和最短路径等6种经典的图算法。这些算法的代码实现，目的和重点在于通用性。如果要获得最佳性能，可以参考其实现进行修改和扩展满足业务需求。另外，研读这些代码，也是理解GraphX编程最佳实践的好方法。

GraphX在淘宝

图谱体检平台

基本上，所有的关系都可以从图的角度来看待和处理，但到底一个关系的价值多大？健康与否？适合用于什么场景？很多时候是靠运营人员和产品经理凭直觉来判断和评估的。如何将各种图的指标精细化和规范化，对于产品和运营的构思进行数据上的预研指导，提供科学决策的依据，是图谱体检平台设计的初衷和出发点。

基于这样的出发点，借助GraphX丰富的接口和工具包，针对淘宝内部林林总总的图业务需求，我们开发一个图谱体检平台。目前主要进行下列指标的检查。

度分布：这是一个图最基础和重要的指标。度分布检测的目的，主要是了解图中“超级节点”的个数和规模，以及所有节点度的分布曲线。超级节点的存在对各种传播算法都会有重大的影响（不论是正面助力还是反面阻力），因此要预先对这些数据量有个预估。借助GraphX最基本的图信息接口degrees: VertexRDD[Int]（包括inDegrees和outDegrees），这个指标可以轻松计算出来，并进行各种各样的统计。

二跳邻居数：对大部分社交关系来说，只获得一跳的度分布远远不够，另一个重要的指标是二跳邻居数。例如，无秘App中好友的好友的秘密，传播范围更广，信息量更丰富。因此，二跳邻居数的统计是图谱体检中很重要的一个指标。对于二跳邻居的计算，GraphX没有给出现成的接口，需要自己设

计和开发。目前使用的方法是：第一次遍历，所有点向邻居点传播一个带自身ID，生命值为2的消息；第二次遍历，所有点将收到的消息向邻居点再转发一次，生命值为1；最终统计所有点上，接收到的生命值为1的ID，并进行分组汇总，得到所有点的二跳邻居。

值得注意的是，进行这个计算之前，需要借助度分布将图中的超级节点去掉，不纳入二跳邻居数的计算。否则，这些超级节点会在第一轮传播后收到过多的消息而爆掉，同时它们参与计算，会影响与它们有一跳邻居关系的顶点，导致不能得到真正有效的二跳邻居数。

连通图：检测连通图的目的是弄清一个图有几个连通部分及每个连通部分有多少顶点。这样可以将一个大图分割为多个小图，并去掉零碎的连通部分，从而可以在多个小子图上进行更加精细的操作。目前，GraphX提供了ConnectedComponents和StronglyConnected-Components算法，使用它们可以快速计算出相应的连通图。连通图可以进一步演化成社区发现算法，而该算法优劣的评判标准之一，是计算模块的Q值，来查看所谓的modularity情况。

更多的指标，例如Triangle Count和K-Core，无论是借助GraphX已有的函数，还是自己从头开发，都陆续在进行中。目前这个图谱体检平台已初具规模，通过平台的建立和推广，图相关的产品和业务逐渐走上“无数据，不讨论，用指标来预估效果”的数据化运营之路，有效提高沟通效率，为各种图相关的业务开发走上科学化和系统化之路做好准备。

多图合并工具

在图谱体检平台的基础上，我们可以了解到各种关系的特点。每种关系都有自己的强项和弱项，例如有些关系图谱连通性好些，而有些关系图谱的社交性好些，因此往往要使用关系A来丰富关系B。为此，借助GraphX，我们在图谱体检平台上开发了一个多图合并工具，提供类似图并集的概念，可快速对指定的两个不同的关系图谱进行合并，产生一个新的关系图谱。

以用基于A关系的图来扩充基于B关系的图，生成扩充图C为例，融合算法的基本思路如图3：若图B中某边的两个顶点都在图A中，则将该边加入C图（如BD边）；若图B中某边的一个顶点在图A中，另一个顶点不在，则将该边和另一顶点都加上（如CE边和E点）；若图A中某边的两个顶点都不在图B中，则舍弃这条边和顶点（如EF边）。

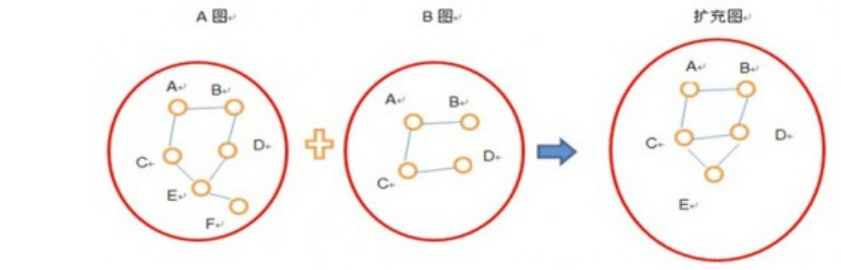

图3 图合并算法

使用GraphX的outerJoinVertices等图运算符，能很简单地完成上述操作。另外，在考虑图合并时，也可为不同图的边加上不同的权重，综合考虑点之间不同关系的重要性。新产生的图，会再进行一轮图谱体检。通过前后三个图各个体检指标的对比，有助于对业务上线后的效果做预估和判断。如果不符合期望，可尝试重新选择扩充方案。

能量传播模型

加权网络上的能量传播是经典的图模型之一，可用于用户信誉度预测。模型的思路是：物以类聚，人以群分。常和信誉度高的用户进行交易的，信誉度自然较高；常和信誉度差的用户有业务来往的，信誉度自然较低。模型不复杂，但淘宝全网有上亿的用户点和几十亿关系边，要对如此规模的巨型图进行能量传播，并对边的权重进行精细的调节，对图计算框架的性能和功能都是巨大的考验。借助GraphX，我们在这两点之间取得了平衡，成功实现了该模型。

流程如图4所示，先生成以用户为点、买卖关系为边的巨型图initGraph，对选出的种子用户，分别赋予相同的初始正负能量值（trustRank & badRank），然后进行两轮随机游走，一轮好种子传播正能量（tr），一轮坏种子传播负能量（br），然后正负能量相减得到finalRank，根据finalRank判断用户的好坏。边的初始传播强度是0.85，这时AUC很低，需要再给每条边，带上一个由多个特征（交易次数、金额……）组成的组合权重。每个特征，都有不同的独立权重和偏移量。通过使用

partialDerivativeAUC方法，在训练集上计算AUC，然后对AUC求偏导，得到每个关系维度的独立权重和偏移量，生成新的权重调节器（WeightAdjustor），对图上所有边上的权重更新，再进行新一轮大迭代，这样一直到AUC稳定时，终止计算。

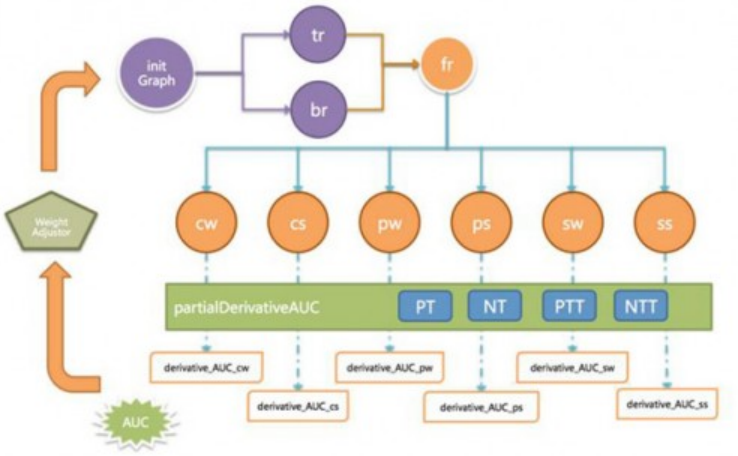

图4 能量传播和训练模型

在接近全量的数据上进行3轮大迭代，每轮2+6次Pregel，每次Pregel大约30次小迭代后，最终的AUC从0.6提升到0.9，达到了不错的用户预测准确率。训练时长在6个小时左右，无论在性能还是准确率上，都超越业务方的期望。

未来图计算的前景

经过半年多的尝试，之前一些想做但因为没有足够的计算能力而不能实现的图模型，现已不是问题。我们将会进一步将越来越多的图模型，在GraphX上实现。这些模型应用于用户网络的社区发现、用户影响力、能量传播、标签传播等，可以提升用户黏性和活跃度；而应用到推荐领域的标签推理、人群划分、年龄段预测、商品交易时序跳转，则可以提升推荐的丰富度和准确性。

本文作者：黄明，花名明风，淘宝网数据挖掘和计算团队负责人，在分布式计算和机器学习算法领域有所涉猎，**Spark**早期研究和布道者之一。新浪微博：**@**明风**Andy**

吴炜，曾在盛大、**MediaV**任职，长期从事数据挖掘方面工作。目前供职于淘宝技术部数据挖掘与计算组。新浪微博：**@**吴炜_数据挖掘机器学习

有什么感想，你也来说说吧！

发表评论

最新评论 最热评论

请您注意

·自觉遵守：爱国、守法、自律、真实、文明的原则

·尊重网上道德，遵守《全国人大常委会关于维护互联网安全的决定》及中华人民共和国其他各项有关法律法规

·严禁发表危害国家安全，破坏民族团结、国家宗教政策和社会稳定，含侮辱、诽谤、教唆、淫秽等内容的作品

·承担一切因您的行为而直接或间接导致的民事或刑事法律责任

·您在CSDN新闻评论发表的作品，CSDN有权在网站内保留、转载、引用或者删除

·参与本评论即表明您已经阅读并接受上述条款