Broadview
www.broadview.com.cn

大数据丛书
十二五国家重点图书出版规划项目

BIG DATA

大数据日知录
架构与算法

张俊林 著

全面梳理大数据相关技术，从数据、算法、策略、应用和系统架构等多个维度进行剖析，既包罗万象，又深入浅出。

《大数据》《信息检索导论》译者王斌 机器学习专家张栋 新浪微博平台及大数据总经理刘子正
盛大文学首席数据官陈运文 CSDN/《程序员》创始人蒋涛
联袂力荐

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

内 容 简 介

大数据是当前最为流行的热点概念之一，其已由技术名词衍生到对很多行业产生颠覆性影响的社会现象，作为最明确的技术发展趋势之一，基于大数据的各种新型产品必将会对每个人的日常生活产生日益重要的影响。

本书从架构与算法角度全面梳理了大数据存储与处理的相关技术。大数据技术具有涉及的知识点异常众多且正处于快速演进发展过程中等特点，其技术点包括底层的硬件体系结构、相关的基础理论、大规模数据存储系统、分布式架构设计、各种不同应用场景下的差异化系统设计思路、机器学习与数据挖掘并行算法以及层出不穷的新架构、新系统等。本书对众多纷繁芜杂的相关技术文献和系统进行了择优汰劣并系统性地对相关知识分门别类地进行整理和介绍，将大数据相关技术分为大数据基础理论、大数据系统体系结构、大数据存储，以及包含批处理、流式计算、交互式数据分析、图数据库、并行机器学习的架构与算法以及增量计算等技术分支在内的大数据处理等几个大的方向。通过这种体系化的知识梳理与讲解，相信对于读者整体和系统地了解、吸收和掌握相关的优秀技术有极大的帮助与促进作用。

本书的读者对象包括对 NoSQL 系统及大数据处理感兴趣的所有技术人员，以及有志于投身到大数据处理方向从事架构师、算法工程师、数据科学家等相关职业的在校本科生及研究生。

图书在版编目（CIP）数据

大数据日知录：架构与算法 / 张俊林著. —北京：电子工业出版社，2014.9
（大数据丛书）
ISBN 978-7-121-24153-6

Ⅰ. ①大…　Ⅱ. ①张…　Ⅲ. ①数据处理　Ⅳ.①TP274

中国版本图书馆 CIP 数据核字（2014）第 194446 号

策划编辑：付　睿
责任编辑：李利健
特约编辑：顾慧芳
印　　刷：北京京师印务有限公司
装　　订：北京京师印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×1092　1/16　印张：25.25　字数：587 千字
版　　次：2014 年 9 月第 1 版
印　　次：2014 年 9 月第 1 次印刷
印　　数：3000 册　定价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

14 图数据库：架构与算法

董小姐　我也是个复杂的动物
嘴上一句带过 心里却一直重复
董小姐　鼓楼的夜晚时间匆匆
陌生的人　请给我一支兰州

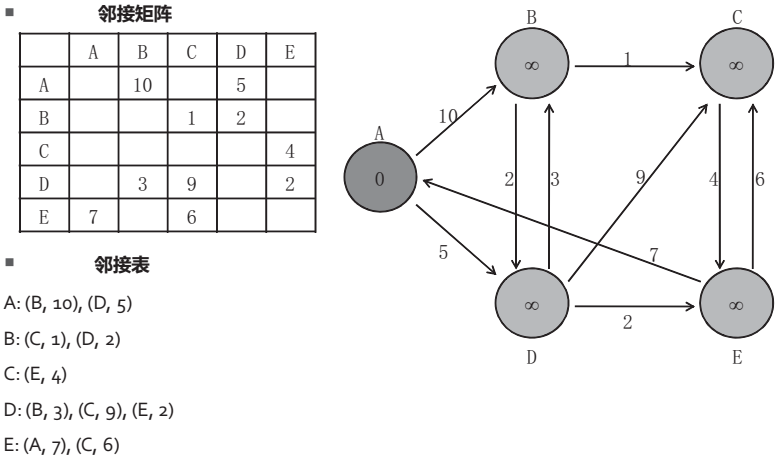
——宋冬野《董小姐》

图计算是一类在实际应用中非常常见的计算类别，当数据规模大到一定程度时，如何对其进行高效计算即成为迫切需要解决的问题。最常见的大规模图数据的例子就是互联网网页数据，网页之间通过链接指向形成规模超过 500 亿节点的巨型网页图。再如，Facebook 社交网络也是规模巨大的图，仅好友关系已经形成超过 10 亿节点、千亿边的巨型图，考虑到 Facebook 正在将所有的实体数据节点都构建成网状结构，其最终形成的巨型网络数据规模可以想见其规模。要处理如此规模的图数据，传统的单机处理方式显然已经无能为力，必须采用由大规模机器集群构成的并行图数据库。

在处理图数据时，其内部存储结构往往采用邻接矩阵或邻接表的方式，图 14-1 是这两种存储方式的简单例子示意图。在大规模并行图数据库场景下，邻接表的方式更加常用，大部分图数据库和处理框架都采用了这一存储结构。

图数据与大数据处理中常见的 KV 数据相比，有自身独有的特点，这也决定了其处理机制与其他类型的海量数据处理系统有很大的差异。具体而言，图数据的数据局部性很差，相互之间有很密切的关联，具体体现就是图节点所展现出的边，其表征着数据之间的关联。很多自然图的结构遵循 Power Law 规则，满足 Power Law 规则的图数据分布极度不均匀，极少的节点通过大量的边和其他众多的节点发生关联。这给分布式存储和计算带来很大的困难，因为数据局部性差意味着数据分布

到集群中的机器时存在潜在的数据分布不均匀或者计算中需要极高的网络通信量等问题。



	A	B	C	D	E
A		10		5	
B			1	2	
C					4
D		3	9		2
E	7		6		

图 14-1　图的表示方式

由于与图相关的应用形式多种多样，为了能够方便地对现有技术进行系统性的介绍，我们将图数据库分为两类：一类是在线查询类，另一类是离线挖掘类。在线查询类图数据库更关注用户查询低延时响应和系统高可用性，比如，Facebook 用户登录时需要将好友列表以及实时变化的信息快速体现到用户交互界面上。离线挖掘类图数据库则更强调数据挖掘等后台处理任务的数据吞吐量及任务完成效率。两者在任务目标上相差很大，这也造成了相关系统设计思路的巨大差异。

本章首先以 Facebook 的 TAO 为例介绍在线查询类图数据库的基本设计思路，之后将重点放在离线挖掘类图数据库上，分别介绍如何对百亿级别的图数据进行数据分片、图计算的计算范型与编程模型，以及若干具有代表性的图数据库系统。

14.1　在线查询类图数据库

14.1.1　三层结构

在线查询类图数据库的主要目的往往是给具体应用提供在线数据读写服务，其中尤其关注数据查询类服务，所以更强调系统的高可用性和读写的低延迟。其体系结构一般由底向上可以划分为三层：分布式存储引擎层、图数据管理层和最上端的图操作 API 层。

为了能够处理海量数据，底层的存储引擎往往采用分布式架构，从理论上看，具体使用何种存储引擎没有限制，实际上，这一层采用 MySQL 数据库居多，主要是可以利用成熟数据库的很多特有功能，比如事务等，这一层只负责数据的存储，分布式管理功能并不在这一层实现。

位于中间层的图数据管理层主要起到以下三个作用。

其一，对底层分布式存储引擎的管理功能，比如，数据的分片与分发、对查询的路由、系统容错等。

其二，图操作逻辑到底层物理存储层读写操作的逻辑转换。图操作逻辑与其他类型的数据操作有较大的差异，比如，经常需要查询图节点之间的关系，为了更好地支持应用，在图数据管理层一般会将数据模型封装成具有图语义的模式，而由于底层存储引擎并不能直接支持这种图语义，所以需要有一个映射和转换过程，比如，若底层存储引擎采用关系数据库，那么需要将图语义转换为对应的若干 SQL 语句，这样才可能实现底层真正存取数据。

其三，在线查询类图数据库往往更注重系统的高可用性和低延迟，其中，读操作的效率更是重中之重。为了达到实时存取的目标，图数据管理层往往会采用优化措施来达到这一目标。

最上层的 API 层主要是封装符合图操作逻辑的对外调用接口函数，以方便应用系统使用在线查询类图数据库。

工业界比较知名的在线查询类图数据库包括 Twitter 的 FlockDB 和 Facebook 的 TAO，这两者的架构基本符合上述三层体系。

FlockDB 是 Twitter 用来存取用户关注关系的图数据库，底层采用 MySQL 作为存储引擎，中间层采用 Gizzard 和 Gizzmo 来进行分布式数据管理，Gizzrad 是用来进行数据分片的开源工具，而 Gizzmo 负责 Gizzard 集群拓扑信息的持久存储以及在 Gizzard 服务器之间传播数据变化信息。两者配合，可以实现数据的分片、数据副本的维护、数据的物理定位和查询路由等分布式管理工作。此外，FlockDB 也提供了方便的对外 API 以利于应用对系统的调用。

本节后续部分重点讲述 Facebook 的 TAO 图数据库，从其实现可以更深切地体会在线查询类图数据库的特性及为满足这些特性所需的设计思路。至于 Neo4j 等常见的在线查询类图数据库，由于其能处理数据的规模有限，所以本章不做专门介绍。

14.1.2　TAO 图数据库

Facebook 是目前世界上最著名的社交网站，如果从数据抽象的角度来看，Facebook 的社交图不仅包括好友之间的关系，还包括人与实体以及实体与实体之间的关系，每个用户、每个页面、每张图片、每个应用、每个地点以及每个评论都可以作为独立的实体，用户喜欢某个页面则建立了用户和页面之间的关系，用户在某个地点签到则建立了用户和地点之间的关系……如果将每个实体看作是图中的节点，实体之间的关系看作是图中的有向边，则 Facebook 的所有数据会构成超过千亿条边的巨型实体图（Entity Graph）。实体图中的关系有些是双向的，比如，朋友关系；有些则是单向的，

比如用户在某个地点签到。同时，实体还具有自己的属性，比如某个用户毕业于斯坦福大学，出生于 1988 年等，这些都是用户实体的属性。图 14-2 是 Facebook 实体图的一个示意片段。

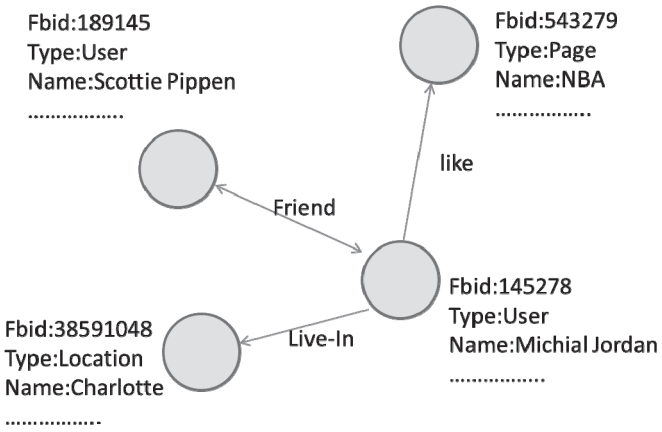


图 14-2　Facebook 实体图（Fbid 是 Facebook 内部唯一的 ID 编号）

Facebook 将所有的实体及其属性、实体关系数据保存在 TAO 图数据库中，网站页面的数据读写请求都由 TAO 来提供服务。TAO 是一个采用数据“最终一致性”的跨数据中心分布式图数据库，由分布在多个数据中心的数千台服务器构成，为了能够实时响应应用请求，TAO 以牺牲强一致性作为代价，系统架构更重视高可用性和低延时，尤其是对读操作做了很多优化，以此保证在极高负载的情况下生成网站页面时的高效率。

TAO 为客户端封装了图操作相关的数据访问 API，使得客户端不仅可以访问实体及其属性，也可以方便地访问各种实体关系数据。比如，对于关系数据的访问可以提供如下关系列表方式的查询接口：

$$(\mathrm{ID}, \mathrm{aType}) \rightarrow [a_{new}, \ldots, a_{old}]$$

其中，ID 代表某个实体的唯一标记，aType 指出关系类型（朋友关系等），关系列表则按照时间先后顺序列出 ID 指向的其他满足 aType 类型关系的实体 ID 列表。例如，（i，COMMENT）就可以列出关于 i 的所有评论信息。

1. TAO 的整体架构

TAO 是跨越多个数据中心的准实时图数据库，其整体架构如图 14-3 所示。首先，TAO 将多个近距离的数据中心组合成一个分区（Region），这样形成多个分区，每个分区内的缓存负责存储所有的实体和关系数据。其中，在一个主分区的数据库和缓存中集中存储原始数据，其他多个从分区存储数据副本（这是一种比较独特的设计方式，建议读者在此处先花些时间考虑一下其设计的出发点，然后阅读后续内容）。

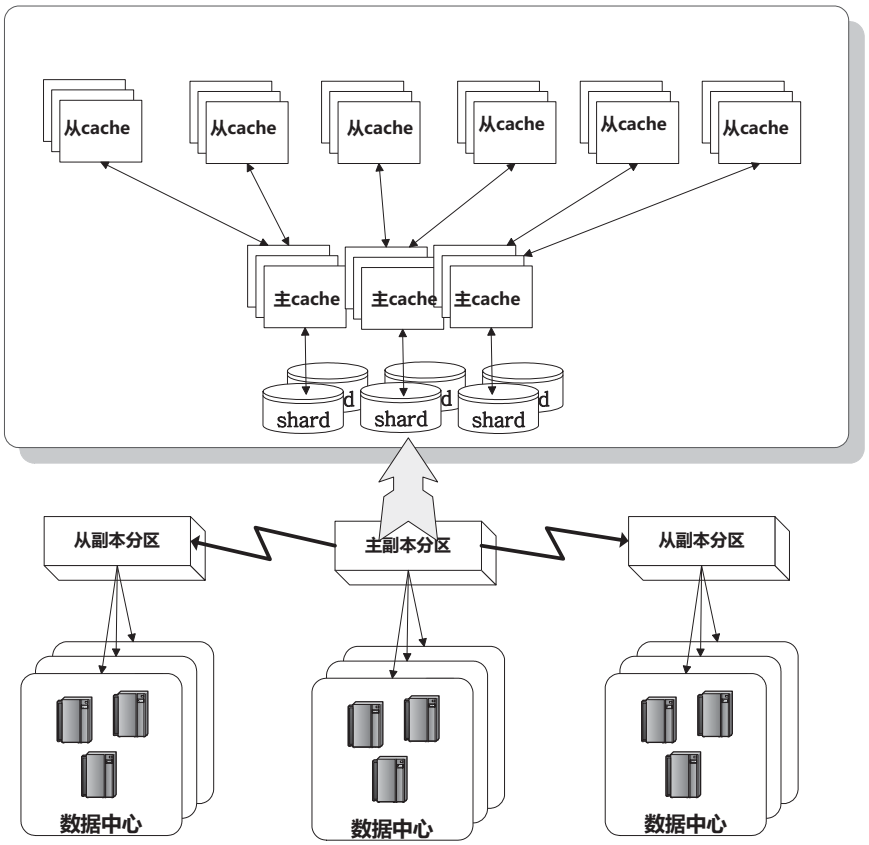


图 14-3　TAO 的跨数据中心架构

之所以如此设计架构，是出于如下考虑：缓存结构是 TAO 中非常重要的一部分，对于快速响应用户读请求有巨大的帮助作用，而缓存需要放在内存中，如果内存资源成本低且足够大，那么理想的情况是每个数据中心都存放完整的数据副本以快速响应用户的读操作，避免用户跨数据中心读取数据这种耗时操作。但是考虑到要存储的数据量太大（PB 级），每个数据中心都分别存储一份完整的备份数据成本过高，所以退而求其次，将在地域上比较接近的多个数据中心作为一个整体来完整地存储所有的备份数据，因为数据中心地域接近，所以通信效率也较高，这样就在成本和效率之间做了一种权衡和折中。

在每个分区会存储完整的实体及其关系数据，TAO 在分区内的存储架构可划分为三层（见图 14-3），底层是 MySQL 数据库层，因为数据量太多，将数据分表后形成若干数据切片（Shard），一个数据切片由一个逻辑关系数据库存储，一台服务器可存储多份数据切片。第二层是与底层数据切片一一对应的缓存层，称之为主 Cache 层（Leader Cache），主 Cache 负责缓存对应的逻辑数据库内容，并和数据库进行读写通信，最上层是从 Cache 层（Follower Cache），多个从 Cache 对应一个主 Cache，负责缓存主 Cache 中的内容。TAO 将缓存设计成二级结构降低了缓存之间的耦合程度，

有利于整个系统的可扩展性，当系统负载增加时，只要添加存储从 Cache 的服务器就能很方便地进行系统扩容。

2. TAO 的读写操作

客户端程序只能与最外层的从 Cache 层进行交互，不能直接和主 Cache 通信（见图 14-4）。客户端有数据请求时，和最近的从 Cache 建立联系，如果是读取操作且从 Cache 中缓存了该数据，则直接返回即可，对于互联网应用来说，读操作比例远远大于写操作，所以从 Cache 可以响应大部分网站负载。

如果从 Cache 没有命中用户请求（Cache Miss），则将其转发给对应的主 Cache，如果主 Cache 也没有命中，则由主 Cache 从数据库中读取，并更新主 Cache（图 14-4 中标 A 和 D 的位置展示了这一逻辑），然后发消息给对应的从 Cache 要求其从主 Cache 加载新数据。

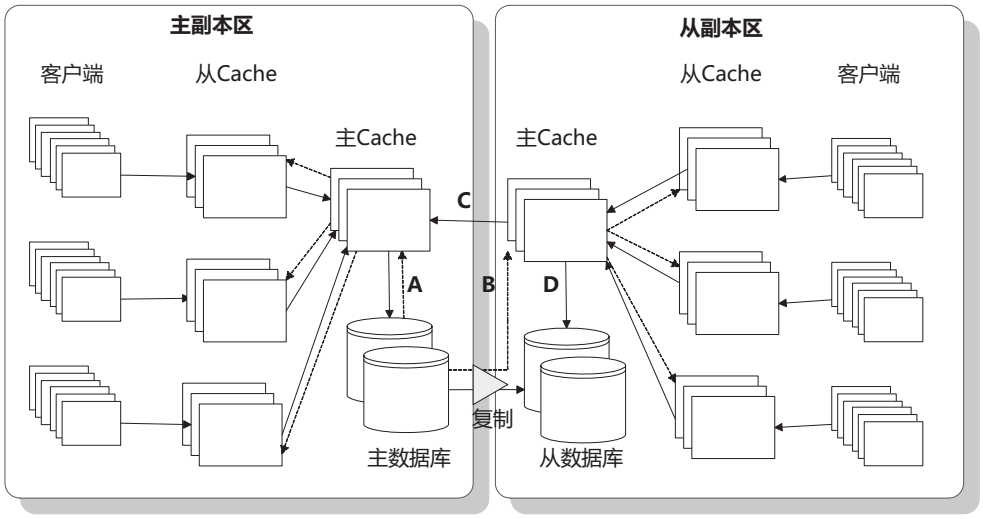


图 14-4　TAO 中的读写操作

对于读取操作，所有的分区不论主从都遵循上述逻辑，但是对于客户端发出的写操作，主分区和从分区的行为有所不同。对于主分区来说，当从 Cache 接收到写操作请求，将其转给对应的主 Cache，主 Cache 负责将其写入对应的逻辑数据库，数据库写操作成功后，主 Cache 向对应的从 Cache 发出消息告知原信息失效或者要求其重新加载。对于从分区来说，当从 Cache 接收到写请求时，将其转给本分区对应的主 Cache，此时主 Cache 并不直接写入本地数据库，而是将这个请求转发到主分区的主 Cache（图 14-4 中标 C 的位置说明了此种情况），由其对主数据库进行写入。

也就是说，对于写操作，不论是主分区还是从分区，一定会交由主分区的主 Cache 来更新主数据库。在主数据库更新成功后，主数据库会通过消息将这一变化通知从分区的从数据库以保持数据

一致性，也会通知从分区的主 Cache 这一变化，并触发主 Cache 通知从分区的从 Cache 更新缓存内容（见图 14-4 标 B 的位置）。

请思考：为何从分区的主 Cache 在读操作未命中时从本地数据库读取，而不是像写操作一样转发到主分区？由本地数据库读取的缺点是很明显的，会带来数据的不一致，因为从数据库可能此时是过期数据，那么这么做的目的何在或者说有何好处？

答案：因为读取数据在 Cache 中无法命中的概率远远大于写操作的数量（在 Facebook 中，大约相差 20 倍），所以跨分区操作对写操作来说，整体效率影响不大，但是如果很多读操作采取跨分区的方法，读取操作效率会大幅降低。TAO 牺牲数据一致性是为了保证读取操作的低延迟。

3. TAO 的数据一致性

TAO 为了优先考虑读操作的效率，在数据一致性方面做出了牺牲，采取了最终一致性而非强一致性。在主数据库有数据变化通知从数据库时，采取了异步通知而非同步通知，即无须从数据库确认更新完成，即可返回客户端对应的请求。所以主数据库和从数据库的数据达到一致有一个时间差，在此期间，可能会导致从分区的客户端读出过期数据，但是经过较小的时延，这种数据变化一定能够体现到所有的从数据库，所以遵循最终一致性。

具体而言，在大多数情况下，TAO 保证了数据的“读你所写”一致性。即发出写操作的客户端一定能够读到更新后的新数值而非过期数据，这在很多情况下是很有必要的，比如，用户删除了某位好友，但如果还能在消息流看到这位好友发出的信息，这是不能容忍的。

TAO 是如何做到这一点的？首先，如果数据更新操作发生在主分区，由上述写入过程可知，一定可以保证“读你所写”一致性，比较棘手的情形是从分区的客户端发出写请求。在这种情形下，从 Cache 将请求转发给主 Cache，主 Cache 将写请求再次转发给主分区的主 Cache，由其写入主数据库，在写入成功后，从分区的主 Cache 通知本分区的从 Cache 更新缓存值，以上操作是同步完成的，尽管此时从分区的数据库可能还未接收到主数据库的更新消息，但是从分区的各级 Cache 已经同步更新了，之后在这个从分区发出的读请求一定可以从各级 Cache 中读到新写入的内容。通过这种手段就可以保证从分区的“读你所写”一致性。

14.2　常见图挖掘问题

本节简述最常见的几个图挖掘问题及算法：PageRank 计算、单源最短路径计算以及二部图最大匹配。后面会有这些问题的具体解决思路及实现代码示例，本节主要讲述问题本身或其典型算法的基本思想。

14.2.1 PageRank 计算

PageRank 是 Google 创始人在 1997 年构建早期搜索系统原型时提出的链接分析算法，自从 Google 在商业上获得空前的成功后，该算法也成为其他搜索引擎和学术界十分关注的计算模型，目前已经衍生并应用在诸多不同的研究领域。

PageRank 计算得出的结果是根据网络拓扑结构分析出的网页重要性评价指标。对某个互联网网页 A 来说，该网页 PageRank 的计算基于以下两个基本假设。

- 数量假设：在 Web 图模型中，如果一个页面节点接收到其他网页指向的入链数量越多，那么这个页面越重要。
- 质量假设：指向页面 A 的入链质量不同，质量高的页面会通过链接向其他页面传递更多的权重。所以质量越高的页面指向 A，则 A 越重要。

通过利用以上两个假设，PageRank 算法刚开始赋予每个网页相同的重要性得分，通过迭代递归计算来更新每个页面节点的 PageRank 得分，直到得分稳定为止。

如将上述问题抽象表示为有向图 $G(V,E)$，其中 V 是所有节点的集合，E 是有向边集合。对图中的某个节点 v_i 来说，有边指向 v_i 的节点集合为 $N_-(v_i)=\{v_j \mid (v_j,v_i)\in E\}$，而节点 v_i 有边指向的节点集合为 $N_+(v_i)=\{v_j \mid (v_i,v_j)\in E\}$，那么每次迭代计算节点 v_i 的 PageRank 得分计算公式如下：

$$PR(v_i;t)=\begin{cases}1/|V| & (t=0)\\ \dfrac{1-d}{|V|}+d\sum_{v_j\in N_-(v_i)}\dfrac{PR(v_j;t-1)}{|N_+(v_j)|} & (t>0)\end{cases}$$

即刚开始时（t=0），所有节点的初始 PageRank 值为 $1/|V|$，$|V|$是图节点总数目；在后续的迭代过程中，v_i 的 PageRank 值是所有指向该节点的边权值之和，而边的权值则是上一轮计算获得的 PageRank 值被所有外出指向边均分的结果。公式中的 d 是调整因子，使得系统可以按照一定概率跳转向图中其他任意节点（即（1-d）/|V|），这是为了防止链接分析中的“链接陷阱”现象出现。

算法不断如上迭代计算，当运行了指定的次数后，或者发现本次计算各个节点的 Pagerank 值与上一次数值大小的差异很小时，则可以中止计算输出最终结果。

14.2.2 单源最短路径（Single Source Shortest Path）

求图节点之间的最短路径是一个经典的图计算问题，现实生活中的很多应用都可以映射到这个问题上，比如，社交网络上的六度空间理论。单源最短路径是众多最短路径问题中的一种：指定源节点 *StartV* 后，求 *StartV* 到图中其他任意节点的最短路径。可形式化将这个问题表述如下：

给定有向图 $G(V,E)$，图中有向边 e:($i\rightarrow j$)上的权值代表从节点 i 到节点 j 的距离，若选定其中

某个节点 *StartV*，求 *StartV* 到 *G* 中其他任意节点的最短距离各是多少。

对于单源最短路径问题来说，需要计算图中其他任意节点到源节点 *StartV* 的最短距离。具体进行分布式计算时的思路为：在迭代计算过程中，每个图节点 *V* 保存当前自己能看到的到 *StartV* 的最短距离，图中的有向边 e:(i→j)上带有权值，权值代表从 i 节点到 j 节点之间的距离。对于任意一个图节点 p 来说，首先从上一轮迭代中有入边指向节点 p 的其他邻接节点中传来的消息中进行查找（每个消息包含指向节点 p 的邻接节点 q 自身目前到 *StartV* 节点的最短距离和 e:(q→p)权值之和），找出外部节点传入的所有距离信息中最短的距离值，如果这个最短距离比节点 p 自身保存的当前最短距离小，则节点 p 更新当前最短的距离，并将这一变化通过出边传播出去，也就是将最新的最短距离加上出边的权值传播给节点 p 有出边指向的节点，代表了从 p 到这个节点最新的最短距离。通过若干次迭代，当节点的数值趋于稳定时，则每个节点最终保存的当前最短距离就是计算结果，即这个节点到 *StartV* 的最短距离。

14.2.3　二部图最大匹配

二部图是这样一个图 $G(V,E)$：其图节点 V 可以划分为两个集合 V_x 和 V_y，图中任意边 $e \in E$ 连接的两个顶点恰好一个在 V_x，一个在 V_y 中。所谓二部图的匹配，指的是二部图 G 的一个子图 M，M 中的任意两条边都不依附于同一个节点。而最大匹配即是指边数最多的那个二部图匹配。

二部图最大匹配也是一个比较典型的图应用问题，比如资源的最优分配与任务的优化安排等问题都可以归结为二部图最大匹配问题。典型的解决二部图最大匹配问题的方法包括网络最大流和匈牙利算法等，在此不做详细叙述，在后文介绍 Pregel 应用时给出了采用随机匹配的方式迭代求解该问题的一个具体代码示例。

14.3　离线挖掘数据分片

对于海量待挖掘数据，在分布式计算环境下，首先面临的问题就是如何将数据比较均匀地分配到不同的服务器上。对于非图数据来说，这个问题解决起来往往比较直观，因为记录之间独立无关联，所以对数据切分算法没有特别约束，只要机器负载尽可能均衡即可。由于图数据记录之间的强耦合性，如果数据分片不合理，不仅会造成机器之间负载不均衡，还会大量增加机器之间的网络通信（见图 14-5），再考虑到图挖掘算法往往具有多轮迭代运行的特性，这样会明显放大数据切片不合理的影响，严重拖慢系统整体的运行效率，所以合理切分图数据对于离线挖掘类型图应用的运行效率来说非常重要，但是这也是至今尚未得到很好解决的一个潜在问题。

对于图数据的切片来说，怎样才是一个合理或者是好的切片方式？其判断标准应该是什么？就像上面的例子所示，衡量图数据切片是否合理主要考虑两个因素：机器负载均衡以及网络通信总量。

如果单独考虑机器负载均衡，那么最好是将图数据尽可能平均地分配到各个服务器上，但是这样不能保证网络通信总量是尽可能少的(参考图 14-5 右端切割方式，负载比较均衡，但是网络通信较多)；如果单独考虑网络通信，那么可以将密集连通子图的所有节点尽可能放到同一台机器上，这样就有效地减少了网络通信量，但是这样很难做到机器之间的负载均衡，某个较大的密集连通子图会导致某台机器高负载。所以，合理的切片方式需要在这两个因素之间找到一个较稳妥的均衡点，以期系统整体性能最优。

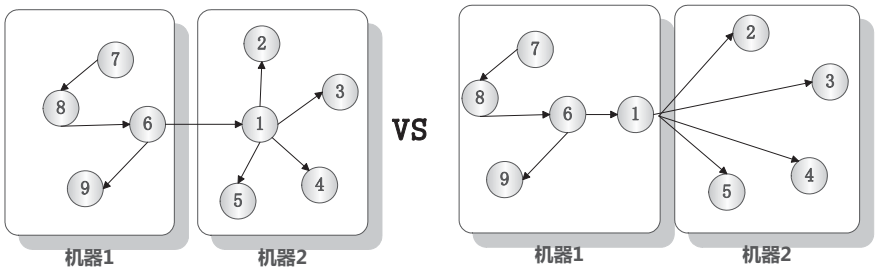


图 14-5　不同图切割方式的差异

下面介绍两类从不同出发点切割图数据的方法，并分别介绍典型的具体切分算法及其对应的数学分析，首先需要强调一点：在选择具体的切分算法时并非越复杂的算法越可能在实际系统中被采纳，读者可以思考其中的道理，在后面会给出解答。

14.3.1　切边法（Edge-Cut）

现在面临的问题是：给定一个巨大的图数据和 p 台机器，如何将其切割成 p 份子图？解决这个图切割问题有两种不同的思路。

切边法代表了最常见的一种思路，切割线只能穿过连接图节点的边，通过对边的切割将完整的图划分为 p 个子图。图 14-6 代表将 7 个节点的图分发到 3 台机器上，左端展示了切边法方式，图节点的编号代表节点被分发到的机器编号。

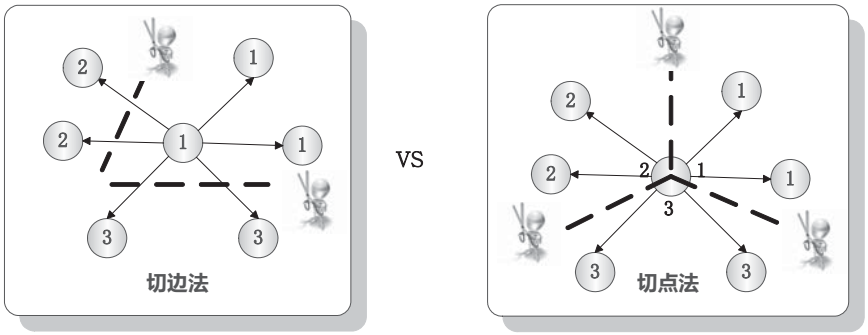


图 14-6　切边法和切点法

通过切边法切割后的图数据，任意一个图节点只会被分发到一台机器，但是被切割开的边数据会在两台机器中都保存，而且被切割开的边在图计算的时候意味着机器间的远程通信。很明显，系统付出的额外存储开销和通信开销取决于被切割开的边的数量，图切割时通过的边越多，则系统需额外承载的存储开销和通信开销越高。

前文有述，衡量图数据分片合理与否有两个考虑因素：负载均衡和机器通信量，所以对于切边法来说，所有具体的切割算法追求的目标不外是：如何在尽可能均衡地将图节点分配到集群中的不同机器上这一约束下，来获得最小化切割边数量。

假设图的节点集合为 V，将任意节点 $v \in V$ 分发到某台机器后，以 $M(v) \in \{1,2,\cdots,p\}$ 代表节点 v 被分发到的机器编号，以：

$$W(e) = \begin{cases} 1 & \text{如果边被切开} \\ 0 & \text{如果边没被切开} \end{cases}$$

代表边被切割后的代价函数。那么，可将优化目标定义为：

$$\min \sum_{e \in E} W(e)$$

$$\text{s.t.} \max_m |\{v \in V \mid M(v) = m\}| < \lambda \frac{|V|}{p}$$

即在每台机器被分发到的节点尽可能均匀的条件约束下，求切割边最少的方法。其中，$|V|/p$ 代表所有的节点被 p 台机器均分所得数值，$\lambda \geq 1$ 代表不平衡调节因子，通过调节λ的大小可以控制节点分配的均匀度，当其值为 1 时，要求完全均分，其值越大，允许的不均衡程度越高。

从上述形式化描述可以看出，λ约等于 1 的时候，这个问题本质上是一个图切割中的均衡 p 路分区（Balanced p-way Partitioning）问题，解决这个问题有很多相关研究（有兴趣的读者可以阅读本章参考文献[4]），但是由于图切割算法的时间复杂度较高，基本不太适合处理大规模数据，所以在真实的大规模数据场景下很少被采用。

在实际的图计算系统中，经常使用的策略是节点随机均分法，即通过哈希函数将节点均分到集群的各个机器中，并不仔细考虑边切割情况。Pregel 和 GraphLab 都采用了这种策略。这种方法的优点是快速、简单且易实现，但是从定理 14.1 可以证明这种方法会将图中绝大多数的边都切开。

【定理 14.1】如果将图中的节点随机分发到集群中的 p 台机器上，被切开的边数比例的数学期望是：

$$\mathbb{E}\left[\frac{|Edge\ Cut|}{|E|}\right] = 1 - \frac{1}{p}$$

证明：对于任意一条边 e 来说，如果联系边的两个图节点被分发到不同的机器上，则边被切开。如果其中一个图节点被分配到某台机器后，另一个图节点被分发到同一台机器的概率为 $1/p$，也即其被切开的概率为 $1-1/p$。因此可证。■

由定理 14.1 可知，假设集群包含 10 台机器，则被切割的边比例大约为 90%，即 90%的边会被切开，而如果包含 100 台机器，则 99%的边会被切开。可见，这种切分方式是效率很低的一种。

14.3.2 切点法（Vertex-Cut）

切点法代表另外一种切割图的不同思路。与切边法不同，切点法在切割图的时候，切割线只能通过图节点而非边，被切割线切割的图节点可能同时出现在多个被切割后的子图中。图 14-6 右侧是切点法示意图，从图中可看出，图中心的节点被切割成三份，也就是意味着这个节点会同时出现在被切割后的三个子图中。

与切边法正好相反，切点法切割后的图中，每条边只会被分发到一台机器上，不会重复存储，但是被切割的节点会被重复存储在多台机器中，因此，同样存在额外存储开销。另外，如此切割带来的问题是：图算法在迭代过程中往往会不断更新图节点的值，因为某个节点可能存储在多台机器中，也即存在数据多副本问题，所以必须解决图节点值数据的一致性问题。对这个问题，在后面讲解 PowerGraph 系统时，会给出一种典型的解决方案。

那么，既然切点法图中的边都没有被切割，机器之间是否就无须通信开销了呢？事实并非如此，在维护被切割的图节点值数据一致性时仍然会产生通信开销。所以，对于切点法来说，所有具体算法追求的合理切分目标是：如何在尽可能均匀地将边数据分发到集群的机器中这个约束条件下，最小化被切割开的图节点数目。

可以进一步形式化表达上述切分目标为：假设图的边集为 E，将任意边 $e \in E$ 分发到某台机器后，以 $A(e) \in \{1,2,\cdots,p\}$ 代表边 e 被分发到的机器编号，以 $A(v) \subseteq \{1,2,\cdots,p\}$ 代表图节点 v 的副本所在的机器集合，$|A(v)|$ 代表其数值。那么，可将优化目标定义为：

$$\min_{A} \frac{1}{|V|} \sum_{v \in V} |A(v)|$$

$$\text{s.t.} \max_{m} |\{e \in E \mid A(e) = \text{m}\}| < \lambda \frac{|E|}{p}$$

即在每台机器被分发到的边尽可能均匀的条件约束下，求平均副本数最少的方法。其中，$|E|/p$ 代表所有边被 p 台机器均分所得数值，$\lambda \geqslant 1$ 代表不平衡调节因子，通过调节λ的大小可以控制边分配的均匀度，当其值为 1 时，要求完全均分，其值越大，允许的不均衡程度越高。

同样，由于采用复杂图切割算法的时间复杂度太高，所以实际系统中最常用的还是边随机均分

法，即通过哈希函数将边均匀地分发到 p 台机器，这基本上是效率最高的一种切割方式，定理 14.2 对这种边随机均分法的数学性质做出了量化描述。另外，如果能够精心选择对边的哈希函数 $h(i \to j)$，我们可以保证每个节点的副本数目不会超过 $2\sqrt{p}$（这里 p 是集群的机器数目），只需构造如下哈希函数即可。

$$h(i \to j) = \sqrt{p} \times (h(i) \bmod \sqrt{p}) + (h(j) \bmod \sqrt{p})$$

即可达到此目的，其中，$h(i)$和 $h(j)$是针对图节点 ID 的均匀哈希函数，同时可以调整机器数目，保证 $\sqrt{p}$ 属于整数。

【定理 14.2】如果将图中的边随机分发到集群中的 p 台机器上，则图节点副本数目的数学期望为：

$$\mathbb{E}\left[\frac{1}{|V|}\sum_{v \in V} |A(v)|\right] = \frac{p}{|V|}\sum_{v \in V}(1-(1-\frac{1}{p})^{D|v|})$$

其中，$D|v|$代表节点 v 的边数。

证明：根据数学期望的性质，可知

$$\mathbb{E}\left[\frac{1}{|V|}\sum_{v \in V} |A(v)|\right] = \frac{1}{|V|}\mathbb{E}[\sum_{v \in V} |A(v)|]$$

对于任意图节点 v 来说，其副本个数的数学期望可以通过考虑与节点 v 相关联的边的随机分发过程来推理。若以指示变量 X_i 来代表第 i 号机器至少分发到与节点 v 相连的一条边这一事件，那么事件 X_i 的数学期望为：

$\mathbb{E}[X_i] = 1 - p$（机器 i 不会被分发到任意与 v 关联的边）

$$= 1 - (1 - \frac{1}{p})^{D|v|}$$

所以，对于图节点 v 来说，其副本数的数学期望是：

$$\mathbb{E}[|A(v)|] = \sum_{i=1}^{p}\mathbb{E}[X_i] = p(1-(1-\frac{1}{p})^{D|v|})$$

即有

$$\mathbb{E}\left[\frac{1}{|V|}\sum_{v \in V} |A(v)|\right] = \frac{1}{|V|}\mathbb{E}[\sum_{v \in V} |A(v)|]$$

$$= \frac{p}{|V|}\sum_{v \in V}(1-(1-\frac{1}{p})^{D|v|})$$

由此可证。■

现实世界中的大多数图的边分布都遵循 power law 法则，理论和实践已经证明，对于遵循这一法则的图数据来说，属于切点法的边随机均分法要比切边法里的节点随机均分法强，其计算效率要高出至少一个数量级。所以总体而言，对于一般情形的图数据，采取切点法要明显优于切边法。

请思考：为何不是越复杂、有效的切分算法越受欢迎？

解答：一般来说，图挖掘算法分为两个阶段。

阶段一：集中式图数据切分与分发；阶段二：分布式图计算。

如果采用复杂的图切割算法，则系统负载均衡好，机器间通信量较少，所以第二阶段运行的效率高，但是采用复杂算法不仅开发成本高，在第一阶段付出的时间成本也很高，甚至因此付出的时间成本要高于在第二阶段产生的效率收益，所以选择何种切分算法也需要有全局的效率权衡。

14.4 离线挖掘计算模型

对于离线挖掘类图计算而言，目前已经涌现出众多各方面表现优秀而各具特点的实际系统，典型的比如 Pregel、Giraph、Hama、PowerGraph、GraphLab、GraphChi 等。通过对这些系统的分析，我们可以归纳出离线挖掘类图计算中一些常见的计算模型。

本节将常见的计算模型分为两类，一类是图编程模型，另一类是图计算范型。编程模型更多地面向图计算系统的应用开发者，而计算范型则是图计算系统开发者需要关心的问题。在本节中，关于编程模型，主要介绍以节点为中心的编程模型及其改进版本的 GAS 编程模型；关于计算范型，则重点介绍同步执行模型和异步执行模型。这几类模型已经被广泛采用在目前的大规模图挖掘系统中。

14.4.1 以节点为中心的编程模型

以节点为中心的编程模型（Vertex-Centered Programming Model）首先由 Pregel 系统提出，之后的绝大多数离线挖掘类大规模图计算系统都采用这个模型作为编程模型。

对图 $G=(V,E)$来说，以节点为中心的编程模型将图节点 $vertex \in V$ 看作计算的中心，应用开发者可以自定义一个与具体应用密切相关的节点更新函数 *Function(vertex)*，这个函数可以获取并改变图节点 *vertex* 及与其有关联的边的权值，甚至可以通过增加和删除边来更改图结构。对于所有图中的节点都执行节点更新函数 *Function(vertex)*来对图的状态（包括节点信息和边信息）进行转换，如此反复迭代进行，直到达到一定的停止标准为止。

典型的图节点更新函数 *Function(vertex)*基本遵循如下逻辑。

伪码：以节点为中心编程模型的节点更新函数

```
Function(vertex) begin
    x[]←read values of in- and out-edges of vertex ;
    vertex.value←f(x[]) ;
    foreach edge of vertex do
        edge.value←g(vertex.value, edge.value);
    end
end
```

即首先从 *vertex* 的入边和出边收集信息，对这些信息经过针对节点权值的函数 *f*()变换后，将计算得到的值更新 *vertex* 的权值，之后以节点的新权值和边原先的权值作为输入，通过针对边的函数 *g*()进行变换，变换后的值用来依次更新边的权值。通过 *vertex* 的节点更新函数，来达到更新部分图状态的目的。

以节点为中心的编程模型有很强的表达能力。研究表明，很多类型的问题都可以通过这个编程模型来进行表达，比如很多图挖掘、数据挖掘、机器学习甚至是线性代数的问题都可以以这种编程模型来获得解决。这也是为何以图节点为中心的编程模型大行其道的根本原因。

14.4.2　GAS 编程模型

GAS 模型可以看作是对以节点为中心的图计算编程模型的一种细粒度改造，通过将计算过程进一步细分来增加计算并发性。GAS 模型明确地将以节点为中心的图计算模型的节点更新函数 *Function(Vertex)* 划分为三个连续的处理阶段：信息收集阶段（Gather）、应用阶段（Apply）和分发阶段（Scatter）。通过这种明确的计算阶段划分，可以使原先的一个完整计算流程细分，这样在计算过程中可以将各个子处理阶段并发执行来进一步增加系统的并发处理性能。

这里假设当前要进行计算的节点是 u，并以此为基础来说明 GAS 模型。

在信息收集阶段，将 u 节点的所有邻接节点和相连的边上的信息通过一个通用累加函数收集起来：

$$\sum \leftarrow \oplus_{v \in Nbr[u]} g(D_u, D_{(u,v)}, D_v)$$

其中，D_u、D_v 和 $D(u,v)$ 分别是节点 u、节点 v 和从 u 到 v 的边上的值信息，$v \in Nbr[u]$ 代表所有有边指向 u 节点的其他节点。用户可以定义累加函数⊕的逻辑，具体定义方式与应用有关，比如，可以是数值累加求和，也可以是对数据求并集操作等。

信息收集阶段计算得到的最终值∑在接下来的应用（Apply）阶段用来更新节点 u 的当前值：

$$D_u^{\text{new}} \leftarrow a(D_u, \sum)$$

在分发阶段，将节点 u 更新后的当前值通过与节点 u 关联的边分发到其他节点：

$$\forall v \in Nbr[u]:(D_{(u,v)}) \leftarrow s(D_u^{\text{new}}, D_{(u,v)}, D_v)$$

通过以上三个阶段的操作，可以定义以图节点为中心的高度抽象的 GAS 计算模型。在 GAS 模型中，节点的入边和出边在信息收集和分发阶段如何使用取决于具体的应用，比如，在 PageRank 计算中，信息收集阶段只考虑入边信息，分发阶段只考虑出边信息，但是在类似于 Facebook 的社交关系图中，如果边表达的语义是朋友关系，那么在信息收集和分发阶段则是所有边的信息都会纳入计算范围。

14.4.3 同步执行模型

同步执行模型是相对于异步执行模型而言的。我们知道，图计算往往需要经过多轮迭代过程，在以节点为中心的图编程模型下，在每轮迭代过程中对图节点会调用用户自定义函数 *Function(vertex)*，这个函数会更改 *vertex* 节点及其对应边的状态，如果节点的这种状态变化在本轮迭代过程中就可以被其他节点看到并使用，也就是说变化立即可见，那么这种模式被称为异步执行模型；如果所有的状态变化只有等到下一轮迭代才可见并允许使用，那么这种模式被称为同步执行模型。采用同步执行模型的系统在迭代过程中或者连续两轮迭代过程之间往往存在一个同步点，同步点的目的在于保证每个节点都已经接受到本轮迭代更新后的状态信息，以保证可以进入下一轮的迭代过程。

在实际的系统中，两种典型的同步执行模型包括 BSP 模型和 MapReduce 模型。关于 BSP 模型的介绍及其与 MapReduce 模型的关系，可以参考本书“机器学习：范型与架构”一章，这里不再赘述。下面介绍图计算中的 MapReduce 计算模型，总体而言，由于很多图挖掘算法带有迭代运行的特点，MapReduce 计算模型并不是十分适合解决此类问题的较佳答案，但是由于 Hadoop 的广泛流行，实际工作中还有一些图计算是采用 MapReduce 机制来进行的。

Mapreduce 计算模型也可以用来进行大规模的图计算，但是其本质上并不适合做这种挖掘类运算。下面主要探讨如何在该框架下解决大规模图计算的问题，并分析其存在的问题和不足。

1. 使用 Mapreduce 进行图计算

使用 MapReduce 框架来针对大规模图数据进行计算的研究工作相对较少，这主要归结于两方面原因：一方面，将传统的图计算映射为 MapReduce 任务相对其他类型的很多任务而言不太直观；另一方面，从某种角度讲，使用该分布计算框架解决图计算任务也并非最适宜的解决方案。

尽管有上述缺点，但很多图算法还是可以转换为 Mapreduce 框架下的计算任务。下面以 PageRank 计算为例讲述如何在该框架下进行图计算。PageRank 的计算原理在前面已有介绍，本节重点分析如何在 Mapreduce 框架下对算法进行改造，使得可以用多机分布方式对大规模图进行运算。

Mapreduce 框架下的输入往往是 key-value 数据对，其中，value 可以是简单类型，比如数值或字符串，也可以是复杂的数据结构，比如数组或者记录等。对于图数据来说，其内部表示方式以邻接表为宜，这样，输入数据的 key 为图节点 ID，对应的 value 为复杂记录，其中记载了邻接表数据、key 节点的 PageRank 值等。

对很多图算法来说，Mapreduce 内部计算过程中的 Shuffle 和 Sort 操作起到类似于通过图中节点出边进行消息传播的效果。从图 14-7 的 PageRank 伪码中可见此技巧的运用。

```
1: class Mapper
2:    method Map(id n, vertex N)
3:       p=N.PageRank/|N.AdjacencyList|
4:       Emit(id n, vertex N)
5:       for all nodeid m in N.AdjacencyList do
6:          Emit(id m, value p)


1: class Reducer
2:    method Reduce(id m, [p1, p2,……])
3:       M = Null
4:       for all p in  [p1, p2,……] do
5:          if IsVertex(p) then
6:             M= p
7:          else
8:           s = s + p
9:       M.PageRank = s
10:   Emit(id m, vertex M)
```

图 14-7　简化版本的 PageRank 伪码（略去了常规计算中的跳转因素）

在该例的 Map 操作中，输入数据的 key 是图节点 ID，value 是图节点数据结构 N，其中包括邻接表 AdjacencyList 信息以及节点对应的当前 PageRank 值。第 3 行代码计算当前节点传播到邻接节点的 PageRank 分值，第 5、6 行代码将这些分值转换为新的 key1-value1，以邻接节点 ID 作为新的 key，而从当前节点传播给邻接节点的分值作为新的 value1。除此之外，还需要将当前节点的节点信息继续保留，以便进行后续的迭代过程，所以第 4 行代码将输入记录本身再次原封不动地传播出去。

通过 MapReduce 内部的 Shuffle 和 Sort 操作，可以将相同 key1 对应的系列 value1 集中到一起，即将 ID 为 key1 的图节点从其他节点传入的 PageRank 部分分值聚合到一起，这起到了类似于消息传播的作用。图 14-7 示例里的 Reduce 操作中，其对应的输入数据包括图节点 ID 以及对应的 PageRank 部分分值列表，伪码第 4 行到第 8 行累积这部分分值形成新的 PageRank 值，同时判断某个 value1 是否是节点信息（第 5 行代码）。第 9 行代码则更新节点信息内的 PageRank 值，而第 10 行代码输出更新后的节点信息。这样就完成了一轮 PageRank 迭代过程，而本次 Reduce 阶段的输出结果可以作为下一轮迭代 Map 阶段的输入数据。如此循环往复，直到满足终止条件，即可输出最终的计算结果。

```
1: class Combiner
2:     method Combine(id m, [p1, p2,……])
3:         M = Null
4:         for all p in  [p1, p2,……] do
5:             if IsVertex(p) then
6:                 Emit(id m,vertex p)
7:             else
8:                 s = s + p
9:         Emit(nid m, value s)
```

图 14-8　Combine 操作伪码

Mapreduce 计算框架在 Map 操作后会通过网络通信将具有相同 key 值的中间结果记录映射到同一台机器上，以满足后续 Reduce 阶段操作的要求。一般情况下，这种网络传输数据量非常大，往往会严重影响计算效率，而 Combine 操作即为减少网络传输以优化效率而提出。Combine 操作在本地机器 Map 操作后，首先将具有相同 key 值的 Map 结果数据 value 部分进行本地聚合，这样本来应该分别传输的项目即被合并，大大减少了网络传输量，加快了计算速度。对于图计算，同样可以采用这种优化手段改善效率，图 14-8 展示了相应的 Combine 操作，其运行流程与 Reduce 操作大体相似，第 4 行到第 8 行代码累加相同 key 的本地 value 数据，第 9 行代码将这种累加数据传播出去，key 保持不变，value 成为聚合数据 s，这样就大量减少了网络传输量。

上面介绍了如何在 Mapreduce 框架下进行 PageRank 计算，很多其他图算法也可用近似的思路处理，其关键点仍然是通过上述的 Shuffle 和 Sort 操作，将同一节点的入边聚合到一起，而 Reduce 操作可以类似例中的部分数值求和，也可能是取边中的 Max/Min 等其他类型的操作，这依据应用各异，但基本思想无较大的区别。

2. MapReduce 在图计算中存在的问题

MapReduce 尽管已经成为主流的分布式计算模型，但有其适用范围，对于大量的机器学习数据挖掘类科学计算和图挖掘算法来说，使用 Mapreduce 模型尽管经过变换也可以得到解决，但往往并非解决此类问题的最佳技术方案。根本原因在于：很多科学计算或者图算法内在机制上需要进行多轮反复迭代，而如果采用 Mapreduce 模型，每一次迭代过程中产生的中间结果都需要反复在 Map 阶段写入本地磁盘，在 Reduce 阶段写入 GFS/HDFS 文件系统中，下一轮迭代一般是在上一轮迭代的计算结果的基础上继续进行，这样需要再次将其加载入内存，计算得出新的中间结果后仍然写入本地文件系统以及 GFS/HDFS 文件系统中。如此反复，且不必要的磁盘输入/输出严重影响计算效率。除此之外，每次迭代都需要对任务重新进行初始化等任务管理开销也非常影响效率。

下面以 Mapreduce 模型计算图的单源最短路径的具体应用实例来说明此问题的严重性。所谓“单源最短路径”，就是对于图结构 $G<N,E>$（N 为图节点集合，E 为图中边集合且边具有权值，这个权值代表两个节点间的距离），如果给定初始节点 V，需要计算图中该节点到其他任意节点的最短距离是多少。这个例子的图结构如图 14-9 所示，图的内部表示采用邻接表方案。假设从源节点 A 出发，

求其他节点到节点 A 的最短距离，在初始化阶段，设置源节点 A 的最短距离为 0，其他节点的最短距离为 *INF*（足够大的数值）。

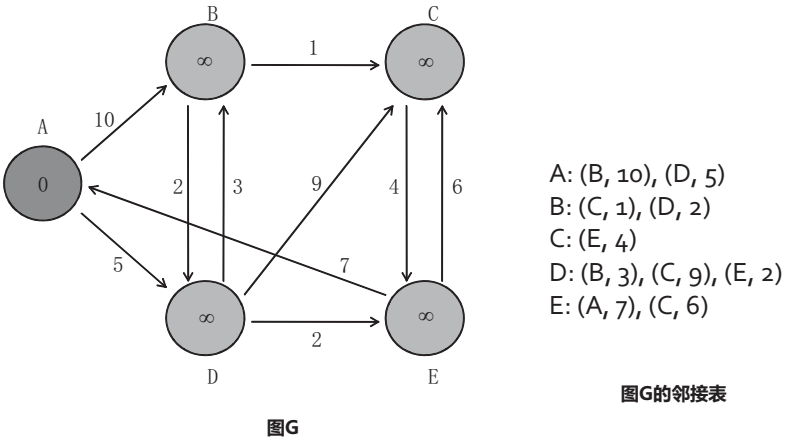


图 14-9　例图及其邻接表

对 Mapreduce 模型来说，计算分为两个阶段，即 Map 阶段和 Reduce 阶段。针对上述问题，Map 阶段的最初输入即为稍加改造的图 G 的邻接表，除了节点的邻接表信息外，还需要额外记载节点当前获得的最小距离数值。以常见的 key-value 方式表示为：key=节点 ID，value=<节点到源节点 A 的当前最小距离 *Dist*，邻接表>。以源节点 A 为例，其 Map 阶段的输入为：<A, <0, <(B, 10), (D, 5)>>>，其他节点输入数据形式与此类似。

Map 阶段对输入数据的转换逻辑为：计算 key 节点的邻接表中节点到源节点 A 的当前最短距离。即将 key-value 转换为 key1-value1 序列，这里 key1 是 key 节点的邻接表中节点 ID，value1 为 key1 节点到源节点 A 的当前最短距离。以源节点 A 为例，其输入为<A, <0, <(B, 10), (D, 5)>>>，经过 Map 转换后，得到输出<B,10>和<D,5>，<B,10>的含义是：B 节点到 A 节点的当前最短距离是 10（由 A 节点到源节点 A 距离 0 加上 B 节点到 A 节点距离 10 获得），<D,5>的含义与之类似。通过此步可以完成 Map 阶段计算，图 14-10 展示了原始输入转换为 Map 阶段输出结果对应的 KV 数值。

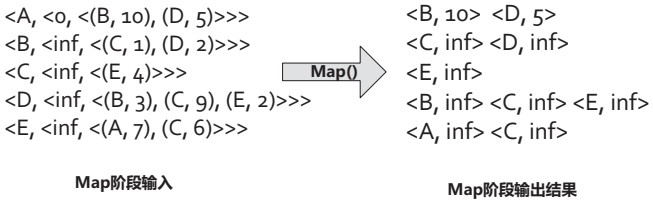


图 14-10　单源最短路径 Map 阶段

在 Map 阶段产生结果后，系统会将临时结果写入本地磁盘文件中，以作为 Reduce 阶段的输入数据。Reduce 阶段的逻辑为：对某个节点来说，从众多本节点到源节点 A 的距离中选择最短的距离作为当前值。以节点 B 为例，从图 14-10 中 Map 阶段的输出可以看出，以 B 为 key 有两项：<B,10>和<B,inf>，取其最小值得到新的最短距离为 10，则可输出结果<B,<10,<(C,1),(D,2)>>>。图 14-11 展示了 Reduce 阶段的输出。

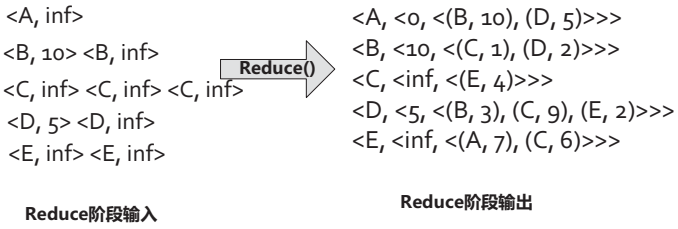


图 14-11　单源最短路径 Reduce 阶段

在 Reduce 阶段结束后，系统将结果写入 GFS/HDFS 文件系统中，这样完成了单源最短路径的一轮计算。使得图节点 B 和图节点 D 的当前最短路径获得了更新。而为了能够获得最终的结果，还需要按照上述方式反复迭代，以本轮 Reduce 输出作为下一轮 Map 阶段的输入。由此可见，如果完成计算，则需要多次将中间结果往文件系统输出，这会严重影响系统效率。这是为何 Mapreduce 框架不适宜做图应用的主要原因。

14.4.4　异步执行模型

异步执行模型相对于同步执行模型而言，因为不需要进行数据同步，而且更新的数据能够在本轮迭代即可被使用，所以算法收敛速度快，系统吞吐量和执行效率都要明显高于同步模型。但是异步模型也有相应的缺点：其很难推断程序的正确性。因为其数据更新立即生效，所以节点的不同执行顺序很可能会导致不同的运行结果，尤其是对图节点并发更新计算的时候，还可能产生争用状况（Race Condition）和数据不一致的问题，所以其在系统实现的时候必须考虑如何避免这些问题，系统实现机制较同步模型复杂。

下面以 GraphLab 为例讲解异步执行模型的数据一致性问题，GraphLab 比较适合应用于机器学习领域的非自然图计算情形，比如马尔科夫随机场（MRF）、随机梯度下降算法（SGD）等机器学习算法。

在讲解异步模型的数据一致性问题前，先来了解一下 GraphLab 论文提出的图节点的作用域（Scope）概念。对于图 G 中的某个节点 v 来说，其作用域 S_v 包括：节点 v 本身、与节点 v 关联的所

有边，以及节点 v 的所有邻接图节点。之所以定义图节点的作用域，是因为在以节点为中心的编程模型中，作用域体现了节点更新函数 $f(v)$能够涉及的图对象范围及与其绑定的数据。

在并发的异步执行模型下，可以定义三类不同强度的数据一致性条件（见图 14-12），根据其一致性限制条件的强度，由强到弱分别为：完全一致性（Full Consistency）、边一致性（Edge Consistency）和节点一致性（Vertex Consistency）。

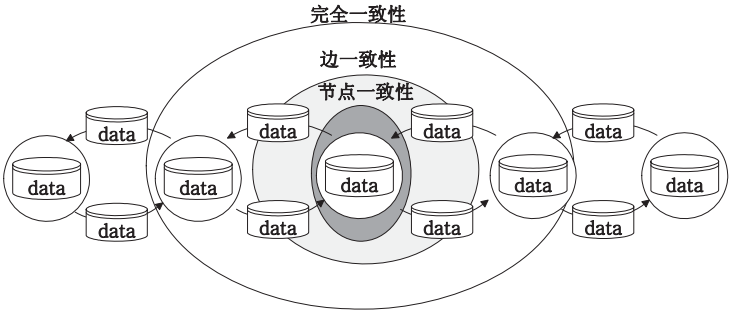


图 14-12　三种一致性

完全一致性的含义是：在节点 v 的节点更新函数 $f(v)$执行期间，保证不会有其他更新函数去读写或者更改节点 v 的作用域 S_v内图对象的数据。因此，满足完全一致性条件的情形下，并行计算只允许出现在无公共邻接点的图节点之间，因为如果两个图节点有公共邻接图节点，那么两者的作用域必有交集，若两者并发执行，可能会发生争用状况，而这违反了完全一致性的定义。

比完全一致性稍弱些的是边一致性条件，其含义为：在节点 v 的节点更新函数 $f(v)$执行期间，保证不会有其他更新函数去读写或者更改节点 v，以及与其邻接的所有边的数据。即与完全一致性条件相比，放松了条件，允许读写与节点 v 邻接的其他图节点的数据。在满足边一致性条件下，并行计算允许出现在无公共边的图节点之间，因为只要两个节点 u 和 v 不存在共享边，则一定会满足边一致性条件。

更弱一些的是节点一致性，其含义为：在节点 v 的节点更新函数 $f(v)$执行期间，保证不会有其他更新函数去读写或者更改节点 v 的数据。很明显，最弱的节点一致性能够允许最大程度的并发，之所以说其限制条件较弱，是因为除非应用逻辑可以保证节点更新函数 $f(v)$只读写节点本身的数据，否则很易发生争用状况，使得程序运行结果不一致。

选择不同的一致性模型对于并行程序执行的结果正确性有很大影响，所谓并行执行的结果正确性，可以用其和顺序执行相比是否一致来进行判断。因此，可以定义“序列一致性”如下：

如果对所有可能的并发执行顺序总是存在与序列执行完全一致的执行结果，在此种情形下，我

们可以将这个并发程序称为是满足序列一致性的。

是否满足序列一致性可以帮助我们验证将一个顺序执行的程序改造为并行执行程序后的正确性。在并行的异步图计算环境下，以下三种情形是可以满足序列一致性的。

情形一：满足完全一致性条件。

情形二：满足边一致性条件，并且节点更新函数 $f(v)$不会修改邻接节点的数据。

情形三：满足节点一致性条件，并且节点更新函数 $f(v)$只会读写节点本身的数据。

上面三种情形可供应用者在设计算法时参考，以在并发性和结果正确性之间做好权衡：一致性条件越弱，则并发能力越强，但是争用状况发生概率越高，即结果可能越难保障正确性。如果应用能够明确节点更新函数的数据涉及范围，就可以根据上述几种情形来进行选择，更好地做到在保证结果正确性的前提下提高并发性能。

14.5 离线挖掘图数据库

从前面已经知道，离线挖掘类图计算范型包括同步模型和异步模型。本节介绍具有代表性的离线挖掘图数据库，其中，Pregel 和 Giraph 采用了典型的同步模型，GraphChi 采用了典型的异步模型，而 PowerGraph 则可以看成是混合模型的代表，即其既可以模拟同步模型，也可以模型异步模型。

14.5.1 Pregel

Pregel 是 Google 提出的大规模分布式图计算平台，专门用来解决网页链接分析、社交数据挖掘等实际应用中涉及的大规模分布式图计算问题。

1. 计算模型

Pregel 在概念模型上遵循 BSP 模型，整个计算过程由若干顺序执行的超级步（Super Step）组成，系统从一个“超级步”迈向下一个“超级步”，直到达到算法的终止条件（见图 14-13）。

Pregel 在编程模型上遵循以图节点为中心的模式，在超级步 S 中，每个图节点可以汇总从超级步 S-1 中其他节点传递过来的消息，改变图节点自身的状态，并向其他节点发送消息，这些消息经过同步后，会在超级步 S+1 中被其他节点接收并做出处理。用户只需要自定义一个针对图节点的计算函数 *F(vertex)*，用来实现上述的图节点计算功能，至于其他的任务，比如任务分配、任务管理、系统容错等都交由 Pregel 系统来实现。

典型的 Pregel 计算由图信息输入、图初始化操作，以及由全局同步点分割开的连续执行的超级步组成，最后可将计算结果进行输出。

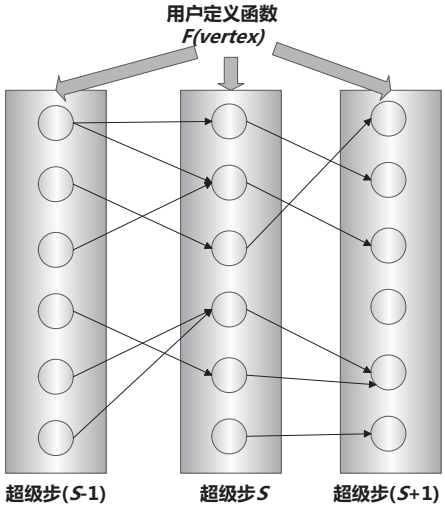


图 14-13　Pregel 的计算模型

每个节点有两种状态：活跃与不活跃，刚开始计算的时候，每个节点都处于活跃状态，随着计算的进行，某些节点完成计算任务转为不活跃状态，如果处于不活跃状态的节点接收到新的消息，则再次转为活跃，如果图中所有的节点都处于不活跃状态，则计算任务完成，Pregel 输出计算结果。

下面以一个具体的计算任务来作为 Pregel 图计算模型的实例进行介绍，这个任务要求将图中节点的最大值传播给图中所有的其他节点，图 14-14 是其示意图，图中的实线箭头表明了图的链接关系，而图中节点内的数值代表了节点的当前数值，图中虚线代表了不同超级步之间的消息传递关系，同时，带有斜纹标记的图节点是不活跃节点。

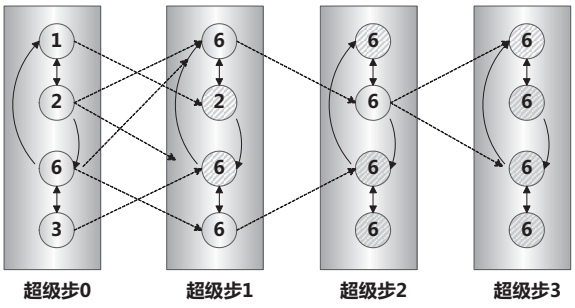


图 14-14　Pregel 传播最大值

从图中可以看出，数值 6 是图中的最大值，在第 0 步超级步中，所有的节点都是活跃的，系统执行用户函数 *F*(*vertex*)：节点将自身的数值通过链接关系传播出去，接收到消息的节点选择其中的最大值，并和自身的数值进行比较，如果比自身的数值大，则更新为新的数值，如果不比自身的数

值大，则转为不活跃状态。

在第 0 个超级步中，每个节点都将自身的数值通过链接传播出去，系统进入第 1 个超级步，执行 *F*(*vertex*)函数，第一行和第四行的节点因为接收到了比自身数值大的数值，所以更新为新的数值 6。第二行和第三行的节点没有接收到比自身数值大的数，所以转为不活跃状态。在执行完函数后，处于活跃状态的节点再次发出消息，系统进入第 2 个超级步，第二行节点本来处于不活跃状态，因为接收到新消息，所以更新数值到 6，重新处于活跃状态，而其他节点都进入了不活跃状态。Pregel 进入第 3 个超级步，所有的节点处于不活跃状态，所以计算任务结束，这样就完成了整个任务，最大数值通过 4 个超级步传递给图中所有其他的节点。算法 14.1 是体现这一过程的 Pregel C++代码。

算法 14.1[C++]：最大值传播

```
Class MaxFindVertex
: public Vertex<double, void, double> {
public:
virtual void Compute(MessageIterator* msgs) {
    intcurrMax = GetValue();                    //节点保存了到目前为止看到的最大值
    SendMessageToAllNeighbors(currMax);         //将当前的最大值通过消息传播出去

    /*从当前消息队列查找是否有比当前值更大的值*/
    for ( ; !msgs->Done(); msgs->Next()) {
        if (msgs->Value() >currMax)
            currMax = msgs->Value();
    }

    if (currMax>GetValue())                     //如果有，更新最大值
        *MutableValue() = currMax;
    else                                        //如果没有，转入休眠态
        VoteToHalt();
    }
};
```

2. 系统架构

Pregel 采用了“主从结构”来实现整体功能，图 14-15 是其架构图，其中一台服务器充当“主控服务器”，负责整个图结构的任务切分，采用“切边法”将其切割成子图（Hash(*ID*)=*ID* mod *n* ，*n* 是工作服务器个数），并把任务分配给众多的“工作服务器”，“主控服务器”命令“工作服务器”进行每一个超级步的计算，并进行障碍点同步和收集计算结果。“主控服务器”只进行系统管理工作，不负责具体的图计算。

每台“工作服务器”负责维护分配给自己的子图节点和边的状态信息，在运算的最初阶段，将所有的图节点状态置为活跃状态，对于目前处于活跃状态的节点依次调用用户定义函数 *F*(*Vertex*)。需要说明的是，所有的数据都是加载到内存进行计算的。除此之外，“工作服务器”还管理本机子

图和其他“工作服务器”所维护子图之间的通信工作。

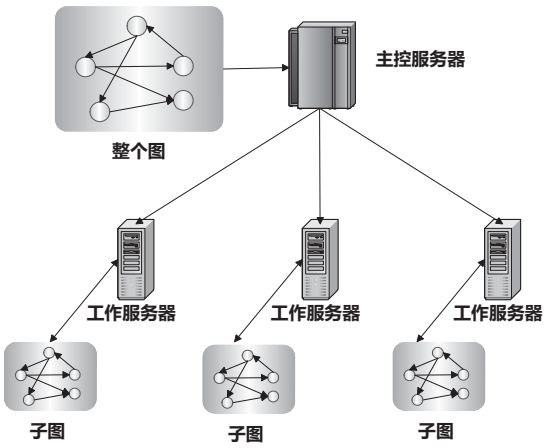


图 14-15　Pregel 的分布协作关系

在后续的计算过程中，“主控服务器”通过命令通知“工作服务器”开始一轮超级步的运算，“工作服务器”依次对活跃节点调用 *F*（*Vertex*），当所有的活跃节点运算完毕，“工作服务器”通知“主控服务器”本轮计算结束后剩余的活跃节点数，直到所有的图节点都处于非活跃状态为止，计算到此结束。

Pregel 采用“检查点”（CheckPoint）作为其容错机制。在超级步开始前，“主控服务器”可以命令“工作服务器”将其负责的数据分片内容写入存储点，内容包括节点值、边值以及节点对应的消息。

“主控服务器”通过心跳监测的方式监控“工作服务器”的状态，当某台“工作服务器”发生故障时，“主控服务器”将其负责的对应数据分片重新分配给其他“工作服务器”，接收重新计算任务的“工作服务器”从存储点读出对应数据分片的最近“检查点”以恢复工作，“检查点”所处的超级步可能比现在系统所处的超级步慢若干步，此时，所有的“工作服务器”回退到与“检查点”一致的超级步重新开始计算。

从上述描述可以看出，Pregel 是一个消息驱动的、遵循以图节点为中心的编程模型的同步图计算框架。考虑到“主控服务器”的功能独特性和物理唯一性，很明显，Pregel 存在单点失效的可能。

请思考：在容错周期选择方面，每一轮超级步都可以进行一次，也可以选择相隔若干超级步进行一次，那么这两种做法各自有何优缺点？

解答：如果选择较短周期的容错措施，在完成任务的过程中，需要的额外开销会较多，但是好

处在于如果机器发生故障，整个系统回退历史较近，有利于任务尽快完成；较长周期的容错措施正好相反，因为频次低，所以平常开销小，但是如果机器发生故障，则需要回退较多的超级步，导致拉长任务的执行过程。所以这里也有一个总体的权衡。

3. Pregel 应用

本节通过若干常见的图计算应用，来说明 Pregel 框架下如何构造具体的应用程序。

（1）PageRank 计算

PageRank 是搜索引擎排序中重要的参考因子，其基本思路和计算原理在本章前面有所说明，此处不再赘述。下面是利用 Pregel 进行 PageRank 计算的 C++示例代码。

算法 14.2[C++]：PageRank

```
Class PageRankVertex
: public Vertex<double, void, double> {
public:
virtual void Compute(MessageIterator* msgs) {
    if (superstep() >= 1) {
    double sum = 0;

    /*根据传入消息更新节点的当前 Pagerank 值*/
    for (; !msgs->done(); msgs->Next())
        sum += msgs->Value();
        *MutableValue() = 0.15 + 0.85 * sum;
    }

    if (supersteps() < 30) { //是否达到指定的迭代次数
        const int64 n = GetOutEdgeIterator().size();
        SendMessageToAllNeighbors(GetValue() / n); //当前 Pagerank 值传播出去
    }
    else {
        VoteToHalt();
        }
    }
};
```

Compute()函数即为前面介绍的针对 *S* 超级步中图节点的计算函数 *F*（*Vertex*），用户通过继承接口类 Vertex 并改写 Compute(MessageIterator* msgs)接口函数，即可快速完成应用开发，其中 MessageIterator* msgs 是 *S*-1 超级步传递给当前节点的消息队列。该计算函数首先累加消息队列中传递给当前节点的部分 PageRank 得分，之后根据计算公式得到图节点当前的 PageRank 得分，如果当前超级步未达循环终止条件 30 次，则继续将新的 PageRank 值通过边传递给邻接节点，否则发出结束通知，使得当前节点转为不活跃状态。

（2）单源最短路径

在图中节点间查找最短的路径是非常常见的图算法。所谓“单源最短路径”，就是指给定初始节点 StartV，计算图中其他任意节点到该节点的最短距离。下面是如何在 Pregel 平台下计算图节点的单源最短路径的 C++代码示例。

算法 14.3[C++]：单源最短路径

```
Class ShortestPathVertex
: public Vertex<int, int, int> {
public:
virtual void Compute(MessageIterator* msgs) {
intminDist = IsSource((vertex_id()) ? 0 : INF;
//中间变量记载当前最短的距离，源节点赋初始值 0，其他赋初始值无穷大

/*从传入消息中查找最短距离*/
for ( ; !msgs->Done(); msgs->Next())
    minDist = min(minDist, msgs->Value());

/*如果传入的最短距离小于目前节点保存的最短距离*/
if (minDist<GetValue()) {
    *MutableValue() = minDist; //更新最短距离
    OutEdgeIteratoriter = GetOutEdgeIterator();

    /*向外传播:当前最短距离+边的权值*/
    for ( ; !iter.Done(); iter.Next())
        SendMessageTo(iter.target(), minDist + iter.GetValue());
}
VoteToHalt();
}
};
```

从代码中可看出，某个图节点 *v* 从之前的超级步中接收到的消息队列中查找目前看到的最短路径，如果这个值比节点 *v* 当前获得的最短路径小，说明找到更短的路径，则更新节点数值为新的最短路径，之后将新值通过邻接节点传播出去，否则将当前节点转换为不活跃状态。在计算完成后，如果某个节点的最短路径仍然标为 *INF*，说明这个节点到源节点之间不存在可达通路。

（3）二部图最大匹配

二部图最大匹配也是经典的图计算问题，下面给出 Pregel 利用随机匹配思想解决该问题的一个思路。

算法 14.4[C++]：二部图最大匹配

```
Class BipartiteMatchingVertex
    : public Vertex<tuple<position, int>, void, boolean> {
```

```
    public:
    virtual void Compute(MessageIterator* msgs) {
        switch (superstep() % 4) {
        /*左端节点*/
        case 0: if (GetValue().first == 'L') {
            SendMessageToAllNeighbors(1);
            VoteToHalt();
        }
        /*右端节点*/
        case 1: if (GetValue().first == 'R') {
          Rand myRand = new Rand(Time());
          for ( ; !msgswitch (supersteps->Done(); msgs->Next()) {
            if (myRan d.nextBoolean()) {
                SendMessageTo (msgs->Source, 1);
                break;
            }
        }
        VoteToHalt();
        }
        /*左端节点*/
        case 2: if (Get Value().first == 'L') {
            Rand myRand = new Rand(Time());
        for ( ; !msgs->Done(); msgs->Next) {
            if (myRand.nextBoolean()) {
              *MutableValue().second = msgs->Source());
              SendMessageTo(msgs->Source(), 1);
              break;
            }
        }
        VoteToHalt(); }
        /*右端节点*/
        case 3: if (GetValue().first == 'R') {
            msgs->Next();
            *MutableValue().second = msgs->Source();
        }
        VoteToHalt();
    }
}};
```

上面的 Pregel 程序采用随机匹配的方式来解决二部图最大匹配问题，每个图节点维护一个二元组：（'L/R'，匹配节点 ID），'L/R'指明节点是二部图中的左端节点还是右端节点，以此作为身份识别标记。二元组的另一维记载匹配上的节点 ID。

算法运行经过以下四个阶段。

阶段一：对于二部图中左端尚未匹配的节点，向其邻接节点发出消息，要求进行匹配，之后转入非活跃状态。

阶段二：对于二部图中右端尚未匹配的节点，从接收到的请求匹配消息中随机选择一个接收，并向接收请求的左端节点发出确认信息，之后主动转入非活跃状态。

阶段三：左端尚未匹配的节点接收到确认信息后，从中选择一个节点接收，写入匹配节点 ID 以表明已经匹配，然后向右端对应的节点发送接收请求的消息。左端节点已经匹配的节点在本阶段不会有任何动作，因为这类节点在第一阶段中根本就没有发送任何消息。

阶段四：右端尚未匹配的节点至多选择一个阶段三发过来的请求，然后写入匹配节点 ID 以表明已经匹配。

通过上述类似于两次握手的四个阶段的不断迭代，即可获得一个二部图最大匹配结果。

14.5.2　Giraph

Giraph 是用于大规模图计算的 Hadoop 开源框架，在计算机制和体系结构上，Giraph 基本上可以看作是 Pregel 的开源版本，绝大部分机制和架构与 Pregel 类似，也是采取 BSP 计算模型，采用以节点为中心的编程模型，并通过消息传递方式将多轮超级步运算串接起来。

但是 Giraph 也有一些技术与 Pregel 有不同之处，本节主要讲述 Giraph 和 Pregel 之间有较大差异的两个技术点。当然，除此之外，Giraph 还有其他一些技术改进，比如，虽然数据默认是全部放入内存进行计算，但是用户也可以选择使用外存存储数据，此改进可以使系统不受集群内存总量限制，能够运算超出内存限制的海量数据。考虑到其他差异点的影响较小，所以此处不再赘述，我们重点讲解重大的差异点。

值得强调的重大的差异点有两个：其一，Giraph 解决了 Pregel 存在的单点失效问题；其二，出于技术复用目的，Giraph 底层采用的是 Hadoop 的 HDFS 作为存储系统以及对应的 MapReduce 计算机制。前文有述，使用 MR 机制进行图运算效率很低，那么 Giraph 是如何克服这一效率瓶颈的呢？

1. 单点失效问题

请思考：如果你是 Giraph 的架构师，任务是要解决 Pregel 的“主控服务器”单点失效问题，你会如何做？

在讲解 Pregel 时，我们提到“主控服务器”存在单点失效问题。“主控服务器”承担总体管理和调度的重要角色，如果其发生故障，整个图计算系统都无法正常运行，所以解决这个问题对系统可用性的提升有很大的帮助。

Giraph 采用 ZooKeeper 来解决“主控服务器”单点失效问题。当“主控服务器”发生故障时，ZooKeeper 可以侦测到这一现象，并通过投票从其他“工作服务器”中选举出新的“主控服务器”承

担后续的管理与调度任务。

但是，问题并未获得彻底解决。

我们知道，Pregel 图计算是有状态的，“主控服务器”需要维护系统持续运行所需的状态信息，这些状态信息包括：数据分片和“工作服务器”之间的映射关系、目前正在进行的超级步步数、检查点存储路径信息等。如果“主控服务器”失效，尽管新选出的“主控服务器”可以接管后续的管理调度操作，但是之前的运行状态信息如何获得？答案同样是：ZooKeeper。本书“分布式协调系统”一章有述，ZooKeeper 也可以作为各种配置文件和状态信息的存储地，所以只要将这些系统状态信息存入 ZooKeeper，就可以彻底解决“主控服务器”的单点失效问题。

2. Hadoop 的低效率问题

Giraph 在底层采用 HDFS 作为存储，MapReduce 作为内部计算机制，要避免 MR 运算机制对于迭代类计算的低效率问题，必须采取一定的改造手段。

首先，一个图计算任务在 Giraph 内部就是由一个巨大的 Map-Only 任务构成的，没有 Reduce 阶段。其次，系统将所有计算需要的数据保持在内存中，这样就避免了 MapReduce 固有的磁盘频繁读/写操作。在传统的 MR 计算任务中，Shuffle 起到了消息传递的作用，而在 Giraph 中，串接起各个超级步的消息通信机制则采取 Netty 网络通信框架，Netty 是由 JBOSS 提供的一个 Java 开源框架，它提供异步的、事件驱动的网络应用程序框架和工具，常常用以快速开发高性能、高可靠性的网络服务器和客户端程序。通过以上几个改进措施，Giraph 就可以既重用 Hadoop 框架，又避免了传统 MR 运算机制在迭代计算中的低效率问题。

算法 14.5 列出了“工作服务器”运行的 Map 任务函数计算流程，由此可以看出 Giraph 是如何通过 Map-Only 任务来完成图计算的 BSP 模型的。

算法 14.5[伪码]：Map 任务函数

```
Void Map()
{
    superstep=0;
    vertices=LoadPartitionsFromHDFS(path);

    do{
        For(Vertex v: vertices.GetActive()){
            v.compute();
        }

        SendMessagesToWorkers();
        CheckPointIfNecessary(superstep);
        Synchronize();
        superstep++;
```

```
    }while(superstep<MAX_SUPERSTEP&& !allHalted(vertices));

    DumpResultToHDFS(vertices);
}
```

关于 Giraph，除了上面介绍的与 Pregel 的两个较大差异点外，算法 14.6 列出了使用 Giraph 解决单源最短路径问题的简化版代码，读者可与前面提到的 Pregel 对应算法的代码进行对比，可以看出两者基本一致，关于其他常见的图算法，也可仿照“Pregel”一节列出的代码一一写出。

算法 14.6[Java]：单源最短路径

```
public void compute(Iterable<DoubleWritable> messages)
{
    double minDist = Double.MAX_VALUE;
    for (DoubleWritable message : messages) {
        minDist = Math.min(minDist, message.get());
    }
    if (minDist<getValue().get()) {
        setValue(new DoubleWritable(minDist));
        for (Edge<LongWritable, FloatWritable> edge : getEdges()) {
            double distance = minDist + edge.getValue().get();
            sendMessage(edge.getTargetVertexId(),new DoubleWritable(distance));
        }
    }
    voteToHalt();
  }
```

类似于 Giraph，HAMA 也是针对大规模图计算的开源 Hadoop 孵化项目，同样也构建于 HDFS 和 MapReduce 之上，不同点在于 HAMA 除了适合做图计算外，也可以进行科学计算的任务。相较于 Giraph，HAMA 在项目活跃度及开源社区参与人数方面的差距都较大，其发展前景与 Giraph 相比稍逊一筹。

14.5.3　GraphChi

GraphChi 是 GrapLab 实验室推出的单机版大规模图计算系统，虽然没有采取目前常用的分布式架构，但是其运算效率及大规模数据处理能力却毫不逊色于采用并行架构的图计算系统。实验表明，在一台配备 8GB 内存和 256GB 容量 SSD 硬盘，以及 1TB 容量普通硬盘的苹果笔记本电脑上，其大规模图计算的效率与当前主流的分布式架构性能相当。

GraphChi 是如何做到这一点的呢？从其设计方案可以学习到设计大数据处理系统的一些技巧。

1. *并行滑动窗口*（Parallel Sliding Windows，PSW）

为了加快运行效率，目前几乎所有的挖掘类大规模图计算系统都是将数据加载到内存进行计算

的，GraphChi 也不例外，但问题是单机环境的内存有限，不可能将所有的图数据加载到内存，所以必须将图切割成若干子图，每次加载子图数据到内存进行计算，依次加载各个子图可以完成一轮迭代，经过若干轮迭代，即可完成图的计算任务。并行滑动窗口（PSW）即是完成上述流程的具体实现方案。

PSW 在对图进行切分时，虽然看上去是采用切点法，但本质上是采用了切边法。在对图节点进行顺序编号并由小到大排序后（称为技巧 1），根据编号大小，将图节点划分为不相交的 *P* 个间隔（Interval），每个间隔代表子图节点集合 *S*，与每个间隔相对应绑定一份数据分片（Shard），用来记录该子图需要保存的信息，这里记载的是图节点集合 *S* 中的所有入边（Inlink）信息，并按照有向边的源节点序号由小到大对边进行排序后有序存储（称为技巧 2），图切分后的效果如图 14-16 所示。

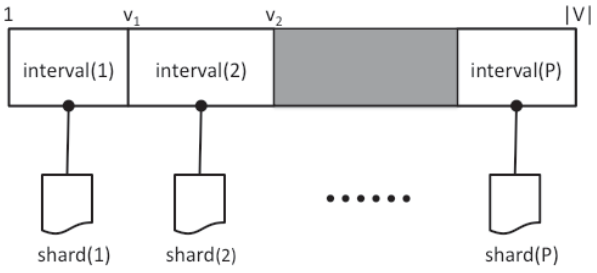


图 14-16　GraphChi 的数据分片

在 PSW 运行的过程中，完成一轮图计算需要依次将每个间隔包含的图节点相关信息加载到内存进行子图数值更新计算。子图数值更新由三个顺序步骤构成：加载子图数据到内存、并发执行图节点更新函数、将更新后的子图数据写回磁盘。

PSW 计算时按顺序依次将每个间隔对应的数据分片加载到内存里，当处理某个间隔 *i* 时，在内存重建间隔 *i* 所包含的子图信息，包括子图包含的所有图节点以及附着在这些节点上的所有边，因为与间隔 *i* 绑定的数据分片 shard(*i*)中记载了子图的所有入边，所以对于一个完整的子图来说，所缺少的只剩下子图的所有出边（Outlink）信息。这些出边信息在哪里保存？它们分布在其他间隔的数据分片中，因为对于间隔 *i* 来说，其包含子图中图节点的所有出边意味着其他间隔中所包含子图的入边，因此需要从其他 *P*-1 个间隔对应的数据分片中将这些出边信息读出，并加载到内存以重建子图。如果这些出边信息不连续地散落在其他间隔对应的数据分片中，则意味着要有很多次的外存随机读操作，这无疑会严重影响系统效率。所幸的是，在经过前面标出的两个实现技巧后，可以保证间隔 *i* 的出边在其他每个间隔对应的数据分片中一定是连续存储的，这样就将潜在的大量磁盘随机读取操作转换成了对磁盘块数据（Block）的 *P*-1 次顺序读取操作，再加上加载间隔 *i* 所需的 1 次数据分片的顺序读操作，以 *P* 次顺序读操作即可将子图计算所需的信息全部加载到内存以重建子图信息，这样无疑会大大提升系统的效率。这是 GraphChi 处理图数据保持高性能的很重要的原因之一。

之所以被称为“并行滑动窗口”，是因为如果当前正在处理第 i 个间隔，则需在其他 P-1 个间隔对应的数据分片中读取对应的连续出边信息，而处理完后继续处理第 i+1 个间隔时，其他 P-1 个间隔对应的数据分片中需要读取的连续出边信息紧跟在之前处理第 i 个间隔时候取的出边信息数据后面，随着计算的进行，每个数据分片就像有一个滑动窗口在随着计算的进行不断下移。图 14-17 展示了其形象示意。

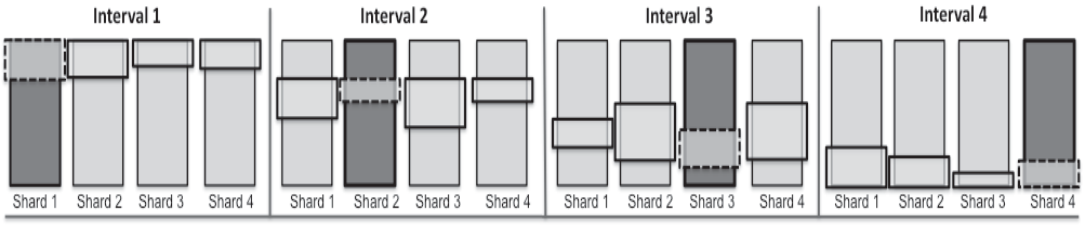


图 14-17　并行滑动窗口示意

当某个间隔的完整子图信息在内存构建完成后，GraphChi 可以并发对子图中的节点执行用户自定义的节点更新函数来完成具体的应用逻辑，所以可以看出，它也属于以节点为中心的编程模型。节点更新函数可以修正边上的权值，因为 GraphChi 是异步模型，所以这种数值变化会立即生效，在并发执行环境下，为了避免不同的节点对同一条边同时更改数值产生的“写-写冲突”，GraphChi 在调度节点并发执行时遵循如下约定。

将子图内的节点分为两类，如果两个节点都在子图中且两者有边相连，那么这些节点作为第一类，子图中其他节点作为第二类。对于第二类节点，可以并发执行，因为可以保证任意两个节点不存在共享边，而对于第一类节点，因为节点之间存在共享边，存在潜在的“写-写冲突”风险，所以顺序执行，以避免这种风险可能带给应用逻辑的结果错误。很明显，第一类节点的计算完全无并发，这是 GraphChi 值得改进的一点，可以引入并发机制来进一步提升系统效率。

由此可见，整个 GraphChi 系统在运行过程中，只有上述对第二类节点的节点更新是并行的，其他阶段（包括间隔的顺序执行），间隔内第一类节点的更新以及后续的更新值写回磁盘都是顺序执行的。

当上述节点的更新阶段完成后，GraphChi 将更新后的边信息写回磁盘。为了提升效率，原先保存在外存的数据块在内存中也顺序放在缓存中，当边值发生更改时，直接更改缓存信息块内对应的数值信息。当整个子图的图节点更新完成后，将更新后的缓存数据块写回各个间隔对应的数据分片的磁盘文件相应的位置，因为数据块是连续的，更新各个文件也以顺序写的方式，所以写磁盘效率非常高，对于一个间隔来说，磁盘顺序写入 P 次即可完成本次迭代的数据输出（称为技巧 3）。更新后的数值在下一个间隔节点更新计算时已经可见，所以 GraphChi 采用的是异步计算模式。

请思考：GraphChi 采用了典型的异步模式，是否有简单的办法将其改造成 BSP 模型等同步计算模式?

答案：只要将 Shard 数据保存两份，一份用来记录本轮迭代之前的数值，在本轮迭代计算时使用这套数据，本轮迭代计算结果写入另一份数据中，这样就可以将其改造为同步计算模式。

完成上述三个阶段即是完成了一个间隔包含的子图内容更新，当序列完成 *P* 个间隔，则完成了一轮迭代过程，经过多轮如此迭代，即可完成图计算任务。整个计算流程如算法 14.7 所示。

算法 14.7[伪码]：PSW 算法

```
foreach iteration do
    shards[]←InitializeShards(P)
    forinterval←1 to P do
        /*加载子图数据：包括 inlink 和 outlink */
        subgraph←LoadSubgraph(interval)
        parallel foreach vertex ∈subgraph:vertex do
            /* 执行用户自定义函数*/
            UDF_updateVertex(vertex)
        end

        /* 将本数据分片写入磁盘*/
        shards[interval].UpdateFully()
        /* 将滑动窗口信息更新到磁盘*/
        for s ∈1,……P, s ≠interval do
            shards[s].UpdateLastWindowToDisk()
        end  //end for
    end  //end for
end  //end foreach
```

为了便于理解上述执行流程，下面以一个具体例子来说明。假设图 G 包含 6 个节点，首先对这些节点一一进行编号，并根据编号大小顺序将其切割为 *P*=3 个间隔，即 1 和 2 号节点作为间隔 1，3 号节点和 4 号节点作为间隔 2，间隔 3 包含 5 号和 6 号节点。每个间隔对应一个数据切片文件，其中存放间隔内子图节点的所有入边，并按照源节点编号由小到大排序。然后根据间隔编号由小到大依次处理子图，示意图 14-18 中阴影部分的边信息代表加载数据分片信息时需要加载到内存的内容，从其顺序执行过程中可以看出滑动窗口的滑动方向。

由上述 GraphChi 的执行流程可以看出其是如何只依靠单机就能快速处理海量图数据的。

之所以能够通过单机处理海量数据，是因为将原始数据进行数据切片后存储在外存，一次读入某个数据切片信息，这样通过调整 *P* 的个数可以保证：尽管数据量很大，但如果外存足够大，只要增加 *P* 的数量，也能够有足够的内存使计算可行。

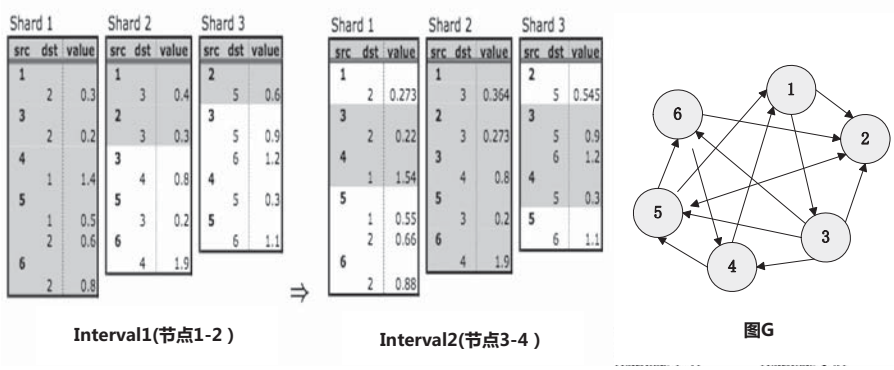


图 14-18　PSW 示例

就像上文所述，整个 GraphChi 在执行期间，并发执行的阶段其实很少，只有在更新节点内容阶段对第二类节点更新的时候才有并发行为，其他各个阶段都是顺序执行的。那么既然这样，为何其运算效率能够达到大规模分布式架构类似的程度呢？其最关键之处即在通过前面介绍的几个实现技巧，将本来不得不进行的大量磁盘随机读写改造成快速的磁盘顺序读写，这里的磁盘 I/O 操作相当于分布式架构中需要进行的机器之间的网络通信阶段的工作，通过这种高效磁盘操作，尽管其对于子图是顺序执行的，也能够大大加快整个系统的运行效率。

2. *应用实例*

下面以 PageRank 计算和寻找连通子图两个简单的例子来说明如何使用 GraphChi 解决具体的应用问题。PageRank 较简单，通过带有注释的代码可以很容易理解其思路。对于寻找连通子图应用来说，其基本思路是：在初始化阶段，将节点 ID 作为图节点值，在之后的多轮迭代过程中，每个节点从邻接节点查找最小值，并将目前看到的最小值通过边传播到邻接节点，当图中节点值稳定后完成计算。标为相同数值的节点集合就是一个连通子图。代码分别如算法 14.8 和算法 14.9 所示。

算法 14.8[C++]：PageRank

```
 void update(graphchi_vertex<VertexDataType,EdgeDataType>&v,graphchi_
context&ginfo){
          float sum=0;
          if(ginfo.iteration==0){
              /* 初次迭代，初始化节点和边值*/
              for(inti=0;i<v.num_outedges();i++){
                  graphchi_edge<float>*edge=v.outedge(i);
                  edge->set_data(1.0/v.num_outedges());
              }
              v.set_data(RANDOMRESETPROB);
          }else{
```

```
            /* 从入边累加邻接节点传入的 pagerank 值 */
            for(inti=0;i<v.num_inedges();i++){
                floatval=v.inedge(i)->get_data();
                sum+=val;
            }

            /* 计算 pagerank 得分 */
            float pagerank=RANDOMRESETPROB+(1-RANDOMRESETPROB)*sum;

            /* 将当前 pagerank 值通过出边传播出去 */
            if(v.num_outedges()>0){
                float pagerankcont=pagerank/v.num_outedges();
                for(inti=0;i<v.num_outedges();i++){
                    graphchi_edge<float>*edge=v.outedge(i);
                    edge->set_data(pagerankcont);
                }
            }

            /* 记录 pagerank 值变化 */
            ginfo.log_change(std::abs(pagerank-v.get_data()));

            /* 设置的节点当前 pagerank 值*/
            v.set_data(pagerank);
        }
    }
```

算法 14.9[C++]：发现连通子图

```
    void update(graphchi_vertex<VertexDataType,EdgeDataType>&vertex,
graphchi_context&gcontext){
        /*初始化，以节点 ID 设置当前值*/
        if(gcontext.iteration==0){
            vertex.set_data(vertex.id());
            gcontext.scheduler->add_task(vertex.id());
        }

        /* 从邻接节点的边中寻找最小值 */
        vid_tcurmin=vertex.get_data();
        for(inti=0;i<vertex.num_edges();i++){
            vid_tnblabel=vertex.edge(i)->get_data();
            if(gcontext.iteration==0)
               nblabel=vertex.edge(i)->vertex_id();

            curmin=std::min(nblabel,curmin);
        }

        /* 设置节点当前值*/
        vertex.set_data(curmin);
```

```
            /*将当前值通过边传播出去*/
            vid_t label=vertex.get_data();

            if(gcontext.iteration>0){
                for(inti=0;i<vertex.num_edges();i++){
                    if(label<vertex.edge(i)->get_data()){
                        vertex.edge(i)->set_data(label);

    /* 将边指向的节点加入调度队列*/
                        gcontext.scheduler->add_task(vertex.edge(i)->vertex_
id());
                    }
                }
            }else if(gcontext.iteration==0){
                for(inti=0;i<vertex.num_outedges();i++){
                    vertex.outedge(i)->set_data(label);
                }
            }
        }
```

14.5.4　PowerGraph

PowerGraph 是一个非常值得关注的系统，其无论在图计算的理论分析方面还是实际的系统实施方面都达到了相当的高度，实际效果也表明 PowerGraph 基本上是目前主流图计算系统里效率最高的。

顾名思义，PowerGraph 主要解决满足以 Power Law 规则分布的自然图的高效计算问题。满足 Power Law 规则的图数据分布极度不均匀，极少的节点通过大量的边和其他众多节点发生关联，比如，Twitter 的关注关系中，仅有占比 1%的图节点与占比 50%的边数据发生关联。这种数据分布的极度不均匀给分布式图计算带来了很多问题，比如很难均匀地切分图数据以及由此带来的机器负载不均衡和大量的机器远程通信，这都严重影响了图计算系统的整体效率。

本节探讨 PowerGraph 是如何对这种分布不均匀的数据进行处理来达到高效率计算目的的。

1. PowerGraph 计算机制

PowerGraph 之所以能够获得极高的运行效率，归结起来，原因在于利用并融合了以下三个因素：切点法分布图数据、利用 GAS 编程模型增加细粒度并发性，以及对中间结果使用增量缓存（Delta Cache）减少计算量。

在本章“离线挖掘数据分片”一节中介绍过切点法，切点法在切割图的时候，切割线只能通过图节点而非边，被切割线切割的图节点可能同时出现在多个被切割后的子图中。在图计算迭代进行过程中，需要更新图节点 u 的数据 D_u，因为切点法导致节点 u 可能会出现在多个机器上，即存在常

见的数据多副本问题，所以这里必须维护多副本的数据一致性。

同时，PowerGraph 采用了细粒度的 GAS 编程模型（请参考本章“离线挖掘计算模型”一节），将对数据更新操作划分为连续的三个阶段，这样可以细粒度地增加并发性。在 GAS 模型下，PowerGraph 是以如下方式保证多副本数据一致性的（见图 14-19）。

对于多副本的节点数据，PowerGraph 会指定其中一份数据为主数据，其他数据作为镜像从数据，在信息收集阶段（Gather），每个副本数据可以并行执行，但是每个副本节点只能累计部分数据，所以在 Gather 阶段完成后，需要进行数据同步操作，从数据将自身累积的那部分信息传到主数据，由主数据进行最后的总累加操作，并采用 Apply 操作更新主数据数值，同时通知其他镜像从数据对节点数据进行更新。在接下来的 Scatter 阶段，各个副本数据同样可以像在 Gather 阶段一样并发执行，去更改邻接边或者邻接节点的数据。可以看出，对于切点法来说，其机器间的通信工作主要是维护数据一致性产生的。

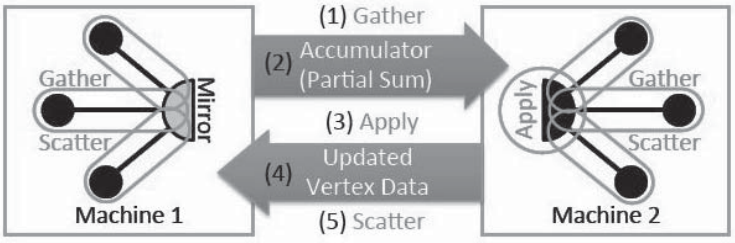


图 14-19　PowerGraph 的数据同步

如上所述，PowerGraph 采用切点法分布图数据，对于符合 Power-Law 原则的自然图，这是比切边法高效的一种数据分布方式，同时 PowerGraph 采用细粒度的 GAS 编程模型，使得 Gather 和 Scatter 两个阶段可以并发操作，这种细粒度的并发模型也可以加快系统的运行效率。对于应用开发者来说，只要实现如下四个接口函数程序实体，即可完成具体的应用，代码如算法 14.10 所示。

算法 14.10[伪码]：GAS 编程接口

```
interface GASVertexProgram(u) {
    // 运行 gather_nbrs(u)
    gather(D_u, D_(u,v), D_v) →Accum

    sum(Accum left, Accum right) →Accum
    apply(D_u, Accum) → D_u^new

    // 运行 scatter_nbrs(u)
    scatter( D_u^new , D_(u,v), D_v) ⇁ ( D_(u,v)^new , Accum)
}
```

其中的 gather、apply 和 scatter 接口即是 GAS 三阶段对应的应用行为，sum 函数定义了如何累加

中间数据，并作为中间函数被 GAS 接口调用，之所以要定义累加函数，是因为不同应用下累加操作的逻辑含义可能不同，比如也许是数值累加行为，也许是对集合采取并集操作等。

当用户遵循 GAS 接口定义清楚上述四个函数的程序实体后，余下的工作交由 PowerGraph 完成，其以节点为中心的编程模型的具体执行语义如算法 14.11 所示。

算法 14.11[伪码]：GAS 编程模型执行语义

```
Input: Center vertex u

/*增量缓存无中间结果，执行 gather 阶段*/
if cached accumulator a_u is empty then
    foreach neighbor v in gather_nbrs(u) do
        a_u← sum(a_u, gather(D_u, D_(u,v), D_v)))
    end
end

D_u←apply(D_u, a_u)

Foreach neighbor v scatter_nbrs(u) do
    (D_(u,v), Δa)← scatter(D_u, D_(u,v), D_v)
    if a_v and Δa are not Empty then
        a_v←sum(a_v, Δa)
    else
        a_v←Empty
end
```

为了能够正确理解上述执行语义，首先需要介绍增量缓存的概念。“增量缓存”也是 PowerGraph 引入的一种增加系统执行效率的重要手段。我们知道，以图为中心的编程模型其实是以与节点有关联的边的数值变化驱动的，Gather 阶段会反复收集边上传递来的数据变化情况，而很多时候，节点上的边数值并未发生变化，此时进行的 Gather 操作其实是一种计算资源的浪费。所以，可以对每个图节点引入一个数值 a_u，并将其放入缓存中，用它来记载上次 Gather 阶段产生的中间结果，如果其值未发生变化，则无须执行 Gather 阶段，通过这种“增量缓存”的方式有效地节省了部分 Gather 阶段的计算，加快了系统执行效率。

为了更容易理解上述执行语义，我们可以结合 PageRank 的例子来进一步梳理，其对应的程序逻辑参考算法 14.12 所示。结合 PageRank 定义的四个操作逻辑，并将这四个操作逻辑分别引入算法 14.11 的 GAS 编程模型执行语义，可以更明晰其执行过程和“增量缓存”的意义所在。

算法 14.12[伪码]：PageRank 算法

```
// gather_nbrs: IN_NBRS
gather(D_u, D_(u,v), D_v):
    return D_v.rank / #outNbrs(v)     //outNbrs(v)是节点 v 的出边个数
```

```
sum(a, b): return a + b

apply(Du, acc):
    rnew = 0.15 + 0.85 * acc
    Du.delta = (rnew - Du.rank)/#outNbrs(u) //Δa 指的是新旧值分配到每条出边的变化量
    Du.rank = rnew

// scatter_nbrs: OUT_NBRS
scatter(Du, D(u,v), Dv):
    if(|Du.delta|>e) Activate(v)
    return delta
```

将 GAS 接口的四个函数引入算法 14.12 后，Gather 和 Apply 阶段很好理解，关键是 Scatter 阶段，在这个阶段，对当前计算的节点 u 的出边指向的节点集合 scatter_nbrs(u)中的任意节点 v 来说，首先调用用户定义的 Scatter 函数，其逻辑是：判断节点 u 的数值变化是否足够大，如果足够大，则需要继续迭代过程，激活节点 v 使得后续计算能够继续，同时返回Δa，也即节点 u 的 PageRank 变化值分配到每条出边后获得的增量值。此时如果“增量缓存”中已经包含了 a_v，则将这种变化累加到 a_v之上，其实这就是在做节点 v 的 Gather 阶段的工作。对于节点 v 来说，如果所有指向它的其他节点都将变化增量值累加到 a_v，那么在本轮计算节点 u 的 Scatter 阶段已经将节点 v 的 Gather 阶段工作做完，当接下来调度到节点 v 执行时，发现节点 v 的 a_v在增量缓存中，所以可以跳过第一阶段直接进入 Apply 阶段。

与 PageRank 算法类似，算法 14.13 列出了单源最短路径的 PowerGraph 接口函数的操作逻辑，读者可以按照上述分析 PageRank 的算法，自行分析其代入执行语义后的执行逻辑。

算法 14.13[伪码]：单源最短路径

```
// gather_nbrs: ALL_NBRS
gather(Du, D(u,v), Dv):
    return Dv+ D(u,v)

sum(a, b): return min(a, b)

apply(Du, new_dist):
    Du= new_dist

// scatter_nbrs: ALL_NBRS
scatter(Du, D(u,v), Dv):
    // 如果值发生变化，则激活邻接节点
    if(changed(Du))
        Activate(v)
    if(increased(Du))
        return NULL
```

```
else
    return D_u+ D_(u,v)
```

2. 模拟同步模型和异步模型

PowerGraph 的整体运行机制为：在最初的阶段，用户可以设置部分节点或者所有的节点为活跃状态，执行引擎负责对每个节点调用用户的 GAS 自定义函数，一旦图节点完成分发阶段（Scatter），则将自身设置为不活跃状态，直到系统中不存在活跃节点，系统执行结束。

GAS 编程模型不仅能够带来更细粒度的并发，其灵活性也可以支持 PowerGraph 同时模拟同步模型和异步模型。其对异步模型的模拟比较直接，只要在数据更新阶段（Apply）和分发阶段（Scatter）使更新数据直接被使用即可。而其对同步模型的模拟可以在 GAS 三个阶段都设立“微同步”（minor step），即所有的节点执行完前面某个子步骤并进行同步后，再依次进入下一个阶段。在所有的节点执行完 GAS 三个阶段后可看作是完成了一次超级步。数据的更新只有在“微同步”完成后才能在下一阶段可见，而本轮被激活为活跃态的节点只有在下一个超级步才开始重新调度。通过这种方式，即可同时模拟同步模型和异步模型，具体选择何种模型可由应用因需而设。

由上所述，可以看出 PowerGraph 能够支持高效图计算的原因，其高效和表达方式的灵活性令人印象深刻，但是通过应用实例也可以看出，编写应用逻辑其实并不如 Pregel 等异步模型那样清晰、直观，学习和使用成本都较高。

参考文献

[1] N. Bronson, Z.Amsden etc. TAO: Facebook’s Distributed Data Store for the Social Graph. In USENIX Association 2013 USENIX Annual Technical Conference. 2013.

[2] Twitter FlockDb. https://github.com/twitter/flockdb.

[3] Neo4j: The graph database, http://www.neo4j.org, 2011.

[4] K. Andreev and H.Räcke. Balanced Graph Partitions. In Proceedings of the 16th SPAA,2004,pp. 120-124.

[5] Y. Low, J. Gonzalez, A.Kyrola, D. Bickson, C.Guestrin, and J.M. Hellerstein, Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. The Proceedings of the VLDB,2012.

[6] G. Malewicz, M.H.Austern, A.J. Bik, J.Dehnert, I.Horn,N.Leiser, and G.Czajkowski, Pregel: a system for large-scale graph processing. In 2010 SIGMOD Conference,2010.

[7] Apache Giraph. http://incubator.apache.org/giraph/.

[8] Apache Hama. http://hama.apache.org/.

[9] A.Kyrola, G.Blelloch, and C.Guestrin, GraphChi: Large-scale graph computation on just a PC. In Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation,2012.

[10] J.E.Gonzalez, Y. Low, H.Gu, D.Bickson, C. Guestrin, Powergraph: Dis- tributed graph-parallel computation on natural graphs. In Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, 2012.