密级:_____

中国科学院大学
University of Chinese Academy of Sciences

硕士学位论文

基于状态更新传播的流式图计算系统设计与实现

作者姓名：段世凯

指导教师: 王伟 副研究员

中国科学院软件研究所

学位类别: 工学硕士

学科专业: 计算机软件与理论

培养单位: 中国科学院软件研究所

2017 年 4 月

Design and Implementation of Streaming Graph Computing Based on State Updating and Spreading

By

Duan Shikai

A Dissertation Submitted to
University of Chinese Academy of Sciences
In partial fulfillment of the requirement
For the degree of
Master of Computer Software and Theory

Institute of Software Chinese Academy of Sciences
April 2017

独创性声明

本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明。

签名：_____　　日期：_____

关于论文使用授权的说明

本人完全了解中国科学院软件研究所有关保留、使用学位论文的规定，即：中国科学院软件研究所有权保留送交论文的复印件，允许论文被查阅和借阅；中国科学院软件研究所可以公布论文的全部或部分内容，可以采用影印、缩印或其它复制手段保存论文。

（保密的论文在解密后应遵守此规定）

签名：_____　导师签名：_____　日期：_____

摘要

图数据结构能够很好的表达数据之间的关联性，因此在社交分析、商品推荐、舆论监测和欺诈检测等应用中被广泛使用。随着互联网的发展，现实社会和生产环境中的图数据越来越呈现海量和动态特性。

目前发展较为成熟的分布式图处理框架 Google Pregel、Spark GraphX 和 GraphLab 等，所处理的图数据都是静态稳定的图数据。针对动态变化图数据的处理，大多集中在算法研究层面上，仅有的 KineoGraph 和 IncGraph 等系统是以串行的方式进行增量式更新，无法充分利用多机并行更新的优势，此外 SpecGraph 提出的基于推测机制的并发更新模型，虽然提高了系统的并行性，但其模型的出发点是状态更新只与顶点接收消息有关而与原始状态无关，这个约束又使得模型的表达能力有限。

基于现有工作的不足，本文提出了一种基于状态更新传播的流式图计算模型，它将连续不断的图数据流抽象成一系列的事件流，将用户关心的图计算结果抽象成图的状态，用户只需要定义图状态如何根据到达的事件增量式地进行状态转换，就能够完成事件流到状态流的映射，提供实时反馈中间计算结果的能力。(1)通过状态直接反应了用户所关注的信息，使得系统无需存储全部的图数据，而只需存储用户关心的数据，从而减少了存储开销；(2)通过采用增量更新和变化传播的方式，使得增量数据对全图的影响范围更小，迭代收敛的速度更快；（3）通过分析典型的图算法特征，抽象出两种常见的状态类型：独立状态和关联状态；（4）通过对独立状态的分布式存储和并发更新策略，以及对关联状态的细粒度分布式锁的更新策略，能够有效解决关联状态下更新冲突的问题，从而提高了系统的并行性和正确率。

实验结果表明，相比较传统的批式图计算系统，实现的基于状态更新传播的流式图计算模型 GraphFlow 系统，能够实时计算并反馈结果，90%的图数据更新请求都能在 12ms 内得到响应；相比较动态图数据的估计模型，GraphFlow 的准确率较高，计算偏差在 1%以内；而采用细粒度分布式锁的方式进行并发更新时，更新冲突的概率在 3%以内；系统的准确率高，实时性好，符合流式图计算的要求。

关键词：图处理系统；实时计算；分布式系统；状态更新；增量图计算；细粒度分布式锁

Abstract

Graph perfectly reflects the association between data, thus it is widely used in applications such as social analytics, commodity recommendation, public opinion monitoring and fraud detection. With the development of the Internet, the graph data becomes huger and increase more rapidly.

While the well-developed distributed graph computing framework like Google Pregel、Spark GraphX and GraphLab focus on the stable graph data, as for dynamic graph data, the main research focus on specific algorithms rather than system or framework. Only KineoGraph and IncGraph apply incremental updating in a serial way instead of parallel updating. In addition, SpecGraph realizes the parallel updating model based on speculative mechanism, but its model assumes the state update is only related to the receiving message and has nothing to do with the original state, this strong assume constraints expressive capability of this model.

Due to the disadvantages of current studies as mentioned above, this paper proposes a streaming graph computing model GraphFlow which is based on state updating and spreading, this model abstracts the continuous change of the graph in a series of events, and abstracts the computing result of the graph into graph state. Users only need to define graph state and how graph state can be transformed according to the current state and arrival event. By this way, the system can complete the event flow to the state flow mapping, and provides real-time feedback of the intermediate state of the streaming graph computing. (1)The graph state which the user define directly reflects the user's concern, making the system only need to store the user defined data instead of the whole graph data, thereby reducing the storage overhead; (2) using the strategy of incremental updates and Change propagation to transform the state of the graph, the sphere of influence of the incremental data is limited within a certain range and the iteration of the graph computing converges faster; (3) summarizing two kinds of common graph state types: independent state and association state; (4) using distributed storage and concurrent update strategy for independent state and fine-grained distributed lock strategy for associated state, the updating conflict problem is solved effectively, thus improving the system parallelism and correct rate.

The experimental result shows that GraphFlow can calculate and feedback the result in real time compared with the traditional batch graph computing system, 90%

graph update request can get the response within 12ms; Compared with the estimated model of the streaming graph computing, the accuracy of GraphFlow is higher, the computing deviation is within 1%; the probability of concurrent update conflict is within 3% by using the fine-grained distributed lock; thus GraphFlow has high accuracy and good real-time performance, which meets the requirements of the streaming graph computing.

Key words: graph processing system; real-time computing; distributed system; state update; incremental graph processing; fine-grained lock

目录

第一章 绪论

1.1 研究背景

图是计算机科学中常用的一类数据结构，它很好地表达了数据之间的关联性。现实世界中有很多数据都可以抽象成图数据，例如 Web 网页之间的链接、社交人物之间的互动以及买卖双方的交易都可以抽象成彼此关联而形成的图。而随着互联网的快速发展，图数据的总量也在急剧增加。如截至 2016 年第四季度，Facebook 包含了 18.6 亿个活跃用户，每个用户平均好友 155 个；Web 链接中图顶点数达到 T 级，边的个数达到 P 级[1]。

因为图数据能够很好地表达数据之间的关联性和聚集情况，因此针对图数据可以挖掘出很多有用信息。比如，通过为购物者之间的关系建模，能很快找到口味相似的用户，并为之推荐商品；在社交网络中，通过传播关系发现意见领袖。图算法及相关的处理框架已经广泛运用在社交分析、商品推荐、舆论监测、欺诈检测等各个领域。

处理这些海量动态的图数据也对现有的图计算模型提出了挑战。一方面，这种超大规模的图数据很难一次性的全部导入内存中进行处理，即使能够借助外存一批一批地处理图数据，这也使得计算延迟显著增加；另一方面，这些图数据又是动态变化、实时更新的，现有的图计算模型要能够在这种动态的数据集上进行增量计算。

现有的成熟的图计算系统如 Google Pregel[2]， Spark GraphX[3]， 这些图计算框架都采用了 BSP[4]（Bulk Synchronized Parallel）模型来处理图数据。然而这些系统都是在静态的图数据上进行的离线批量处理，即每次针对整体的图进行计算，当图发生变化时，需要在变化后的整个图上重新计算一遍。这使得用户等待周期长，无法满足实时计算的要求，也浪费了系统资源[5]。

因此针对图数据不断变化的情况，现有工作提出了很多在动态图上直接进行计算的方法。针对这类动态图计算的问题，大致可以分为两类：估计计算和准确计算。对于估计计算，大部分的算法是希望通过采样来降低时间和空间开销，并通过特定的采样方法来减少和真实值之间的差距。 如 Bar-Yossef Z 等人[6][7][8]研究了在流图上如何通过设计采样规则来估计图中三角形数目；S. Baswana 等人[9][10][11]通过将原始的图数据转化为简单的数据结构来保存图中元素，从而降低了内存消耗。虽然估计算法在一定程度上能够节约内存和计算开销，但其估计的误

差在实际的生产环境中往往变得不可控制，文献[12]指出，针对大体量的无法全部载入内存的图数据，近似算法的错误率在95%-133%之间。对于准确计算，现有的 KineoGraph[13]和 IncGraph[5]提出采用增量计算模型来进行实时计算，然而这种增量式的更新是串行执行的，实时性有限。SpecGraph[14]虽然在上述增量模型的基础上有所改进，提出了基于推测机制的并发更新模型，然而该模型假设顶点的状态只依赖于顶点当前接收的信息，而与顶点之前的旧状态无关，这种假设使得系统的适用性差，很多算法中顶点的状态不仅跟顶点接收消息有关，还跟顶点的旧状态有关，因此模型的表达能力有限。

1.2 论文工作

针对动态图计算的实时性和准确性要求，本文在上述已有的研究基础之上，提出了**基于状态更新传播的流式图计算模型**，能够在原有图状态基础上，并发计算增量信息的影响，而无需在整个图上重新计算，同时通过细粒度分布式锁，实现状态的并发更新，保证计算结果的正确性。本文的主要工作有以下5点：

（1）阐述了图计算领域中图数据**海量**和**动态**特性，图计算**局部性差**和**迭代计算**的特点，并且从**图的划分**、**编程模型**和**计算泛型**三个方面讲解如何针对图数据和图计算的特点，现有工作所提出的关键技术，然后按照所处理的图数据的不同，从批处理图计算和流处理图计算两个方向，总结了现有的图计算模型和系统。

（2）给出了流式图算法的**特征分析**方法。从四个典型的图算法—顶点度分布算法（Degree Distribution，DD），三角形计数算法（Triangle Count，TC），单源点最短路径（Single Source Shortest Path，SSSP），和 PageRank（PR）出发，从影响范围、计算方法、计算顺序、计算特性和计算次数五个维度，分析了它们在流式场景下所呈现的特点。并且给出了常见图算法能够改造成流式图算法所具备的三个特征：①计算方法可以采用增量计算形式；②计算顺序满足序列一致性原则；③计算函数满足代数运算的交换律和结合律。本文所采用的归纳分析方法也适用于其它图算法。

（3）根据流式图算法的特点，建立了**基于状态更新传播的流式图计算模型**。该模型将连续不断的图数据流抽象成一系列的事件流，将图算法的计算结果抽象成图的状态，图在事件的驱动下，根据用户自定义的转换函数，完成了状态的转变，并且实时反馈给用户。这种计算模型有效解决了流式图计算的问题，并且通过增量计算的方式提高了系统的实时性和准确性。

（4）给出了在基于状态更新传播的流式图计算模型上设计算法的思路和步骤。

以 DD、TC、SSSP 和 PR 为例，详细讲解了流式图算法的设计细节。用户可以根据该设计思想将其它符合条件的批式图算法改成流式图算法。

（5）设计并实现了 GraphFlow 系统，从**正确性**、**实时性**和**更新冲突概率**对系统进行评估。结果表明：基于状态更新传播的流式图计算模型构建的算法能够得到较为准确的计算结果，计算偏差在 1%以内；90%的图数据更新请求都能够在 12ms 内得到响应，符合实时性要求；任意两个计算节点更新冲突的概率在 3%以内，并发性好。

1.3 论文组织

本文的后续章节按照如下的方式进行组织：

- 第二章对图计算相关的技术和系统进行了介绍。阐述了图数据和图计算的特点，分析了图计算的关键技术，最后对常见的图计算系统进行了比较。通过本章，用户可以清楚地了解图计算的相关工作。
- 第三章首先以 DD、TC、SSSP、PR 四个图算法为例，分析了流式场景下图算法的特点，并且总结了现有的批处理图算法改造成流式增量图算法所具备的 3 个基本特征，然后详细给出了基于状态更新传播的流式图计算模型的定义和各个组件的说明，最后以连通子图（Connected Components，CC）算法为例介绍了如何使用该模型。通过本章，用户可以清楚了解流式场景下图算法的分析方法，并且了解基于状态更新传播的流式图计算模型的设计思想。
- 第四章在第三章建立的模型基础之上，详细阐述了 DD、TC、SSSP、PR 四个图算法的设计细节，并且给出了算法的伪代码。通过本章，用户可以根据基于状态更新传播的流式图计算模型，来对其它满足要求的图算法进行设计，以满足流式图计算的场景。
- 第五章介绍了采用基于状态更新传播的流式图计算模型的系统 GraphFlow 的设计与实现。从系统架构、模型和算法三个层面，详细阐述 GraphFlow 系统的实现细节。通过本章，用户可以了解整个系统的架构和运行流程。
- 第六章是实验验证和分析。本文从准确性、实时性和更新冲突概率三个层面对 GraphFlow 系统进行了测试。
- 第七章对本文工作做出总结，并且对以后改进的方向提出了几点展望。

第二章 图计算相关工作

在本章中，本文首先分析了图数据和图计算的特点，然后针对这些特点从图的划分、编程模型和计算泛型三个方面阐述了图计算的关键技术，并且总结了常见的图计算系统，最后分析了这些系统的特点以及不足。

2.1 图计算介绍

现实世界中很多情景都可以用图来进行表达。如在社交领域，微博、微信、Facebook 等社交平台中，每个个体可以抽象成图中的一个点，个体之间的关联可以抽象成边，这样形成了超大规模的关系网络图，针对这样的关系网络，可以进行社交分析，为用户推荐好友或者侦测社区；在电商领域，个体对商品的浏览记录和购买记录也可以抽象成图数据，可以根据用户的这些行为记录为他们推荐商品；在万维网中，网页以及网页之间的相互链接可以形成巨大的网页链接图，根据链接信息来对网页的重要性进行排名。由此可见，图的运用是非常广泛的。

图计算相关的问题，可以从数据特性和算法特点两个角度来进行定义和分析。

从数据特性来看，随着信息的传播和互联网的蓬勃发展，这些图数据呈现海量和动态特性。海量是指图数据的体量非常庞大，仅以 Facebook 为例，其在 2016 年第四季度的月用户活跃量已达 18.6 亿，平均每人拥有 155 个好友；动态是指图数据是在不断动态变化的，如 Facebook 在 2010 年第四季度的月用户活跃量在 6 亿左右，6 年时间增加到 18.6 亿，平均增速高达 2 亿/年。图 2-1 展示了 Facebook 每个季度的用户活跃度，可见其增速非常快。

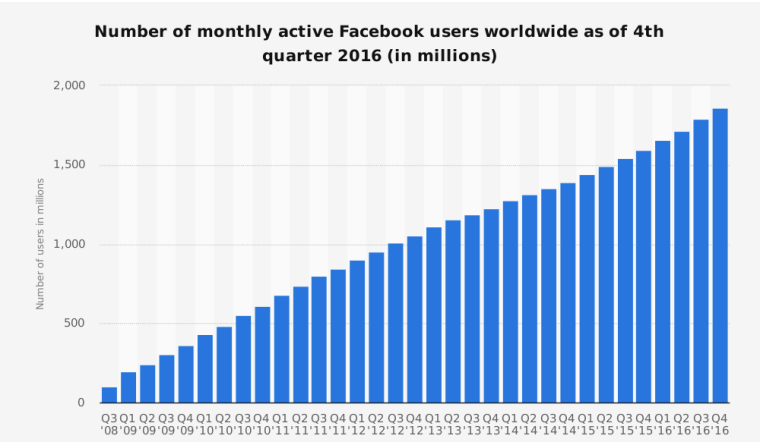


图 2-1 Facebook 月活跃量统计图

数据来源：https://www.statista.com

从算法特点来看，图计算具有局部性差和迭代计算的特点。图结构很好地表达了实体之间的关联，然而现实生活中这种关联往往呈现 Power Law 规则[15]，如图 2-2 所示，满足这种规则的图数据分布极不均匀，只有极少数的顶点的度非常高，大部分顶点的度都非常低。这种极度不均匀的数据给分布式存储和计算带来巨大困难，不仅顶点之间的通信代价提高，而且极度倾斜的数据使得某个计算节点计算压力显著增大。

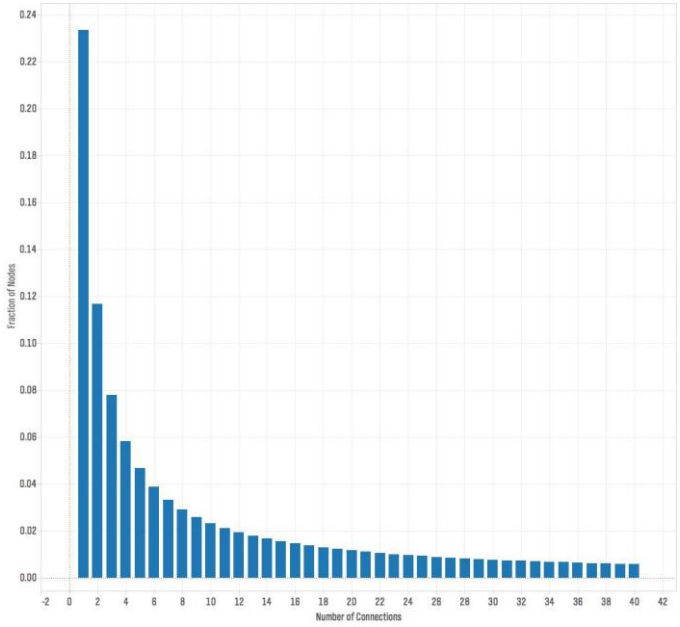


图 2-2 满足 Power-Law 规则的图的顶点的度分布图

数据来源：http://social-dynamics.org/scale-free-network/

针对图数据海量和动态特性，以及图算法局部性差和迭代计算的特点，现有的工作是如何展开的呢？下面将详细介绍图计算相关的关键技术。

2.2 关键技术

图 2-3 从底层细节、模型、算法、系统和应用五个角度分别阐述图计算的关键技术。在本节中本文重点选取了底层细节和模型两个角度来阐述，而算法和系统层面将会在其它章节继续讨论。

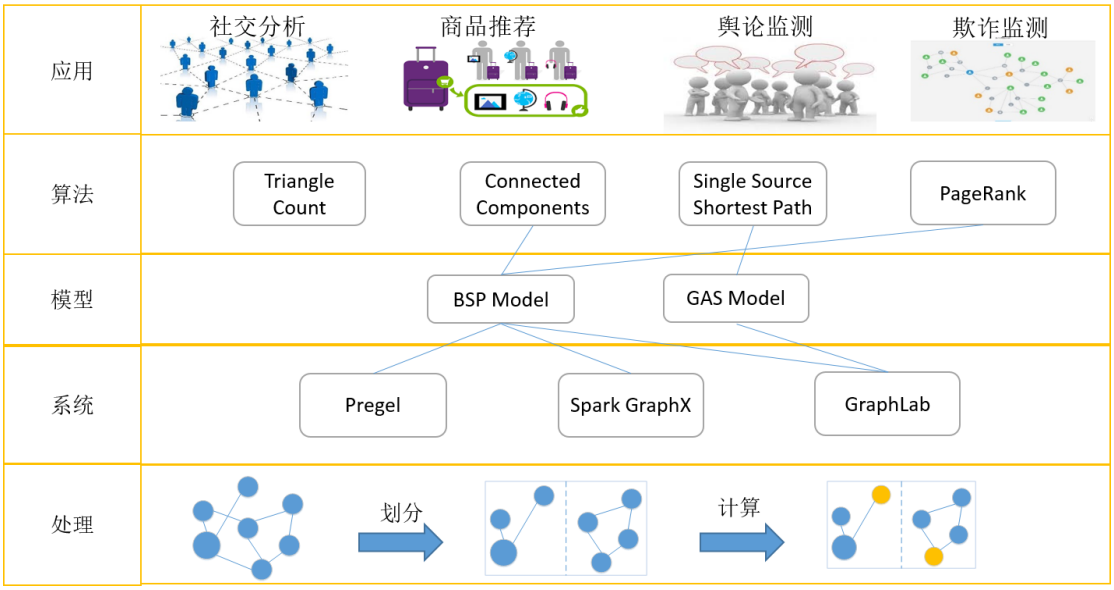


图 2-3 图计算技术概览

从处理方式角度来看，分布式环境下的图计算可以分为两步：划分和计算。即通过某种划分规则，将原来超大规模的图合理的分配到各个计算节点上，然后各个计算节点在该子图上分别进行计算。图的划分在图计算中占据关键位置：一方面这种划分要保证负载均衡；另一方面又希望这种划分能够较好的维持图数据之间的关联性。

从模型角度来看，图计算模型可以分为编程模型和计算泛型[16]：编程模型是面向图计算系统的应用开发者，他们应该按照何种编程模式来编写图的处理逻辑，如常用 Vertex-Centric[17]、Scatter-Gather[18]、Gather-Apply-Scatter[19]都是图计算相关的编程模型；而计算泛型是指图计算引擎采用同步模式还是异步模式来执行计算。

下面，本文将从图的划分、编程模型和计算泛型这三个方面来详细阐述图计算中的关键技术。

2.2.1 图的划分

对于分布式环境下的图计算，首先要解决的问题是如何将海量图数据合理地分配到各个计算节点上。而图数据之间的关联性使得这种划分往往比一般的非图数据更为复杂。其一考虑到计算节点的负载均衡，希望能够将图数据平均分配到每个计算节点；其二考虑到图计算中邻接点的相互通信，希望彼此相邻的顶点能够划分到同一计算节点，以此减少计算节点之间的通信代价。如果划分得不好，不仅会使某个计算节点的计算任务加重，还会使节点之间的通信代价显著提升，而图计算往往又是多轮迭代反复计算的，这使得这种不合理的划分所造成的影响

会显著放大，因此，图的划分是图计算中至关重要的工作。[20][21][22]

现在较为常用的图划分方式有切边法（Edge-Cut）和切点法（Vertex-Cut）。切边法是对边进行切割，然后将这条边的两个顶点分配到不同的计算节点上，如图 2-4 左图所示，通过两条切线将原图分割成 3 个子图，这 3 个子图位于 3 个不同的计算节点上，图中顶点上的编号即为该顶点所在的计算节点编号。切点法是对顶点进行切割，然后将被切割的顶点的多个备份分配到不同的计算节点上，如图 2-3 右图所示，通过将中心顶点切割，将该顶点的 3 个副本分别分配到 3 台不同的计算节点上。由此可见，切边法使得图的边被切割，图的顶点只可能存在集群中的一个计算节点上；而切点法使得图的顶点被切割，但图的边只可能存在集群中的一个计算节点上。

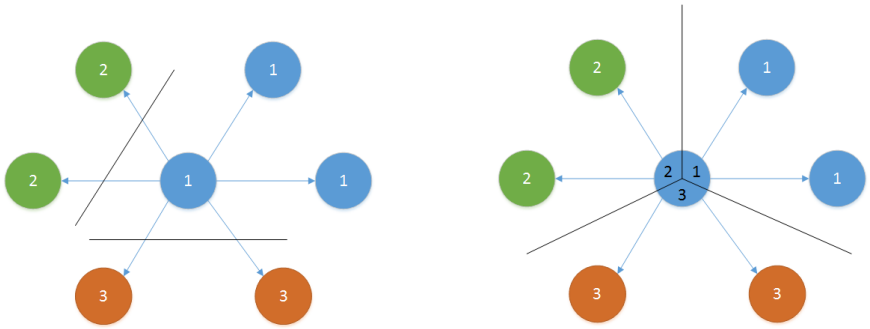


图 2-4 切边法和切点法

根据图数据的特性，可以将图划分问题归为两类：静态图的划分和动态图的划分。

关于静态图划分的问题，学术界很早就展开了相关的研究工作。一种较为简单的方式是散列划分方法[23]，它通过顶点随机哈希的方式，将顶点均匀分散到各个计算节点中，这种划分没有考虑到顶点之间的关联性，因此会产生较高的通信代价，但因为其划分方法简单高效，易于实现，且能够保证负载均衡，被大多数的分布式图计算系统所采用。J Ugander 等人提出了一种基于标签传播的 BLP 算法[24]，该算法在保证负载均衡的情况下，尽量减少边切割的数目，保证了分割数据的局部性。此外还有 Kernighan 和 Lin 提出的基于贪心策略的 KL 算法[25]，G. Karypis 和 V. Kumar 提出的基于多层次图划分的 METIS 算法[26]，这些算法都是以迭代计算的方式，根据特定规则来原图进行划分。

关于动态图划分的问题，现有的研究工作也很多，Ioanna Filippidou 和 Yannis Kotidis[27]提出了一种基于精简生成树结构的图分割算法，它不仅能对任意的演变图进行图分割，还允许不同的应用按需来调整分区；Stanton 和 Kliot[28]提出了一种只依赖于图结构的启发式算法，相对于基于散列的分割方法和 METIS，分割效果有很大提升。另外，Charalampos E.Tsourakakis[29]等人提出了一个新颖的 one-

pass 流图分割算法，该算法统一了两个看似正交的启发式算法：将新到达的顶点放置在具有最大数量邻居结点的分区中或者具有最小数量的非邻居结点的分区中，相对于 METIS，分割时间更短，效果更好。

2.2.2 编程模型

图的编程模型是指图计算框架或系统指导用户按照何种编程规范来编写图计算逻辑，是面向应用开发者的。依据对图的处理视角的不同，可以将图的编程模型划分为通用（General-Purpose）、顶点为中心（Vertex-Centric）和图为中心（Graph-Centric）的编程模型[16]。考虑到最为常用的是以顶点为中心的编程模型，下面本文将重点介绍 Vertex-Centric(以下简称 VC)及其扩展 Gather-Apply-Scatter(以下简称 GAS)。

顶点为中心的编程模型是以顶点为计算单位，采用用户自定义顶点的更新函数 Function(Vertex)来改变顶点的状态，并且根据一定条件有选择性地将这种变化传递给其它顶点。顶点为中心的编程模型具有很强的表达能力，很多如 SSSP 和 PR 等图算法都可以用该模型来进行表示，它也是图计算中最为常用的一种编程模型。

GAS 编程模型可以看成是 VC 编程模型的细粒度的改造。它将原来的顶点更新函数 Function(Vertex)分解为三个阶段：信息收集阶段（Gather），信息处理阶段（Apply）和信息分发阶段（Scatter）。在 Gather 阶段，顶点收集其它顶点（一般是邻接点或者副本顶点）发送过来的信息；在 Apply 阶段，顶点根据 Gather 阶段收集的信息对顶点的状态进行更新；在 Scatter 阶段，顶点会根据 Apply 阶段的更新情况，再将这种更新传给其它顶点（邻接点或副本顶点）。这种细粒度的划分，使得系统能够更大程度地提高不同算子的并发度。在后文的 PowerGraph 系统中会详细阐述该模型是如何工作的。

2.2.3 计算泛型

图的计算泛型是指图的底层执行引擎以何种执行方式来完成图计算的，一般可以为同步计算和异步计算。我们知道图计算往往需要经过多轮的迭代计算，如果在本轮的迭代过程中，顶点的变化能够立即被其它顶点看到并使用，则称这种计算模式为异步计算，相反如果所有顶点的变化只有在下一轮的迭代过程中才能被其它顶点看见和使用，则称这种计算模式为同步计算。[30]

同步计算可以用图 2-5 来表述。图中的竖线为同步点，即在同步点各个线程

强制进行数据同步，同步点之间的过程称为一个迭代阶段，在每个迭代阶段内部，各个线程之间可以独立地执行计算，在每个同步点上，各个线程之间交换彼此计算结果信息，在这样整体同步、计算的迭代过程中推进整个算法的执行。

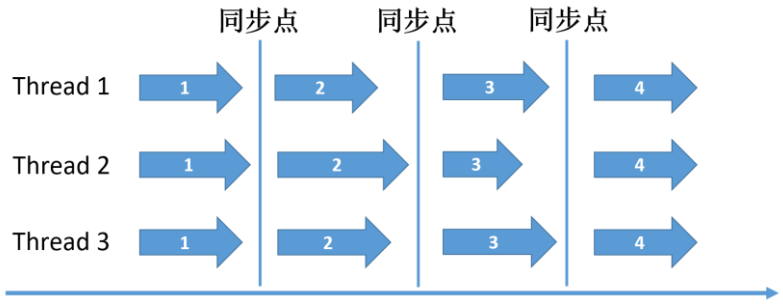


图 2-5 同步计算

BSP 模型和 MapReduce 模型都是典型的同步计算。由于图计算往往是迭代计算，而 MapReduce 模型繁琐的读写磁盘操作会拖慢整个计算速度，因此针对图计算领域，更多的是使用 BSP 模型作为底层的执行引擎。2.3.1 节会详细阐述 BSP 模型在批式图计算领域中的运用。

异步计算可以用图 2-6 来描述。相比较同步计算，异步计算没有显式的同步过程，任意时刻每个并发程序都可以对全局参数进行读取和更新。这样使得整体任务的执行速度加快，但程序的正确性往往无法获得保证。

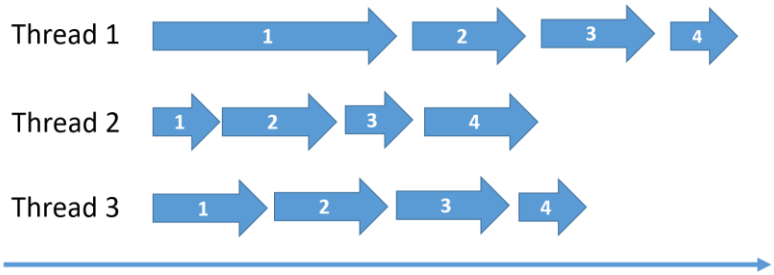


图 2-6 异步计算

2.3 常见图计算系统

现在有诸如 Pregel，Spark GraphX，GraphLab[31]等很多成熟的图计算系统，本文按照所处理图数据的不同，将这些系统分为批式图计算系统和流式图计算系统。批式图计算系统处理的图数据是静态的，而且只有等到全部的图数据处理完毕之后才能反馈计算结果，适合离线图计算情景；而流式图计算系统处理的图数据是动态的，针对动态变化的流式图数据，它能够及时反馈中间计算结果，适合在线图计算情景。

2.3.1 批式图计算系统

图计算一般需要多次迭代，计算过程依赖于顶点之间的通信，而传统的MapReduce 模型更倾向于处理彼此独立的任务，因此其开源实现的 Hadoop 为代表的面向数据并行（Data-Parallel）的计算模型难以对图计算提供高效的支持。[32]

为了解决海量图计算问题，Google 公司提出了基于 BSP(Bulk Synchronous Parallel)思想的大规模分布式图计算平台 Pregel，专门解决网页链接分析、社交数据挖掘等图计算问题。Pregel 使用了 VC 编程模型和同步计算泛型，将整个计算过程分解成由若干个顺序运行的超步（superstep），在每个超步中，活跃的顶点(active vertex)接收上个超步中其它顶点发送过来的消息，并执行用户自定义的计算函数来改变自己的状态，同时将更新的状态再发送给其它顶点，这些消息会在下个超步中被其它顶点接收并处理，然后该顶点进入不活跃状态（inactive vertex）。不活跃的顶点在下个超步中接收到其它顶点的消息会变得活跃，反之如果没有接收其它顶点的消息，将继续保持不活跃的状态，也不会向其它顶点发送消息。超步内各个顶点可以并行处理，而超步之间会对消息进行同步，通过这样以超步为单位的方式迭代运行，直至所有顶点都变得不活跃或没有新的消息产生。用户只需要自己定义超步内顶点的计算逻辑，即可实现计算功能。继 Pregel 之后，一大批以 BSP 为计算模型的分布式图计算系统涌现，如 Spark GraphX，Flink Gelly[33]，Giraph[34]，Hama[35]等，它们都是借助于同步计算和 VC 编程模型，实现了面向静态图数据的图处理系统。BSP 模型的迭代计算过程如图 2-7 所示。

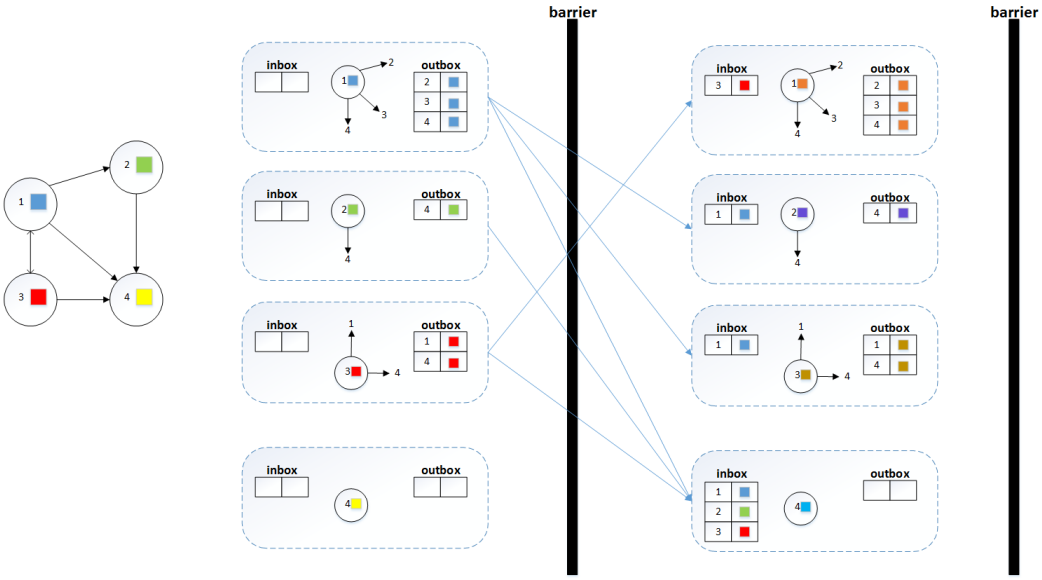


图 2-7 BSP 模型迭代计算过程

鉴于现有的 BSP 计算模型存在通信开销大、短板效应（每轮迭代过程中，计算最慢的节点将拖慢整体计算速度）等问题，卡内基梅隆大学研发出基于异步计

算的 GraphLab 系统。GraphLab 采用分布式共享内存进行顶点通信，以最小化集群计算节点之间的通信量和计算节点上的计算和存储均衡为原则，对图数据进行切分，同时对 VC 编程模型进行扩展，将计算过程抽象为 Gather/Apply/Scatter 三个阶段，这样细粒度的划分能够增加计算的并发性[31]。

考虑到现实世界中的自然图一般符合 Power Law 规则，而这种图的分布极度不均匀，针对这类图数据，卡内基梅隆大学又在 GraphLab 系统的基础上开发了 PowerGraph[36]系统。该系统采用启发式的切点法来划分图数据，同时采用细粒度的 GAS 编程模型向外提供接口，需要注意的是由于 PowerGraph 采用的是点切割，所以一个顶点可能在多台机器上都有副本，PowerGraph 会指定其中一台机器上的顶点为主顶点，其它机器上的顶点为镜像顶点，在 Gather 阶段，各个镜像顶点将自身累积的信息发送给主顶点，主顶点 Apply 阶段更新其值之后，将最新的信息通知给其它镜像顶点。在接下来的 Scatter 阶段，各个镜像顶点可以并发执行，去更改邻接点或邻接边的值。其各个阶段的执行过程如图 2-8 所示。

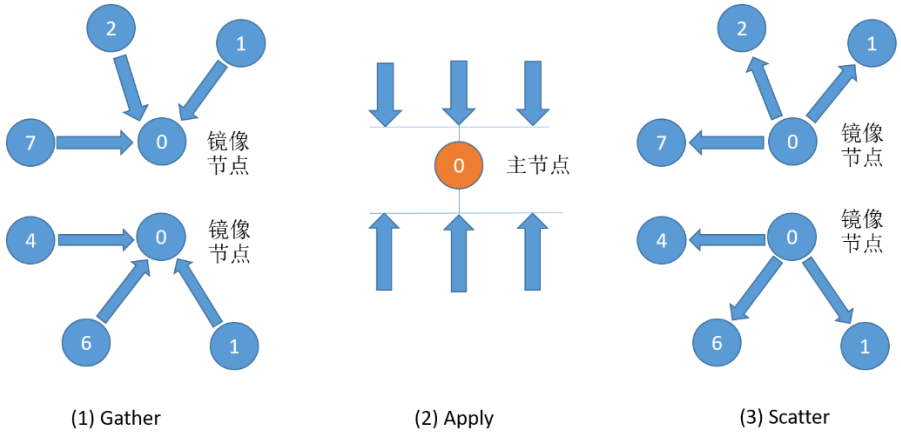


图 2-8 PowerGraph GAS 执行过程图

除了上述的分布式并行图计算系统能够处理静态图数据外，GraphLab 实验室还推出了单机版的 GraphChi[37]也能够处理大规模的图数据。GraphChi 采用了并行滑动窗口（Parallel Sliding Windows, PSW）机制，将原来的大图划分成若干个子图，每次加载子图数据到内存进行计算，依次加载各个子图即可完成一轮迭代，这样经过若干轮迭代，即可完成整个图计算的任务。虽然实验表明 GraphChi 的数据处理能力并不比采用分布式并行计算的系统要差，但本文仅限于讨论分布式情景下的图计算问题，因此不做过多讨论。

2.3.2 流式图计算系统

如 2.3.1 节所述，针对批处理的图系统有很多，而且发展较为成熟，然而这

些系统都是在静态的图数据上进行的离线批量处理，即每次针对整体的图进行计算，当图数据变化时，需要在变化后的整个图上重新计算一遍。这使得用户等待周期长，无法满足实时计算的要求，也浪费了系统资源。因此针对图数据不断变化的情况，现有研究也提出了很多在动态图上直接进行计算的方法。但考虑到图数据和图算法的复杂性，流式图计算系统相比批处理的图系统起步较晚，发展较慢，大多数的研究只针对具体的一个算法或一类算法。因此本节首先分析了流图数据模型和计算模型，最后介绍了支持增量计算的准实时流式图计算系统KineoGraph。

2.3.2.1 流图数据模型

所谓流式图数据是指图的数据（包括图的顶点、图的边、图顶点的值和图边的权重）不再是静态地存储在文件或数据库中，而是以流的形式源源不断地添加到系统中。因此系统中的图是随着时间而动态变化的。依据流中数据的表达形式，现主要有两种典型的流数据模型[38]：

- Cash Register Model：流中的每一项仅仅是数据集中一项，比如在 distinct elements count 中，每一项就是一个数。数据集中的每一项以任意顺序形成数据流。
- Turnstile Model：在该模型中，有一个初始化为空的集合 D，流中的数据由两项组成，一项是数据集的某一项，另一项是一个标志位，可以对集合 D 进行动态改变。如图 2-9 所示，在一个管道中，图的每条边按照一定顺序流入系统中，其中+表示增加一条边，-表示删除一条边，对应这些边的变化，图的结构和状态也在不断变化。在本文中考虑的是边的 Turnstile Model，即图数据流是按照边的添加和删除来进行组织的。

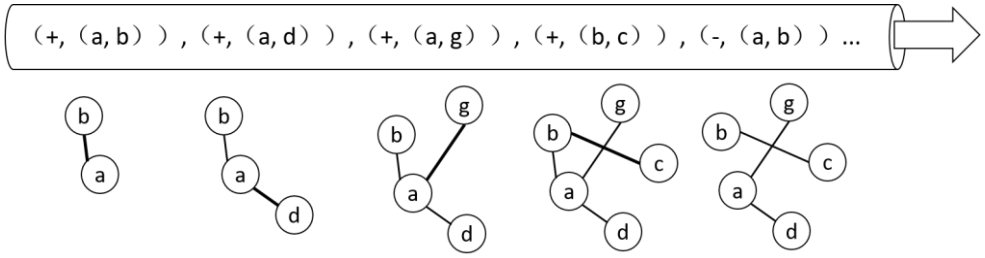


图 2-9 Turnstile Model

2.3.2.2 流图计算模型

针对这类动态图计算的问题，根据计算的准确性，计算模型大致可以分为两类：估计计算和准确计算。

（1）估计计算

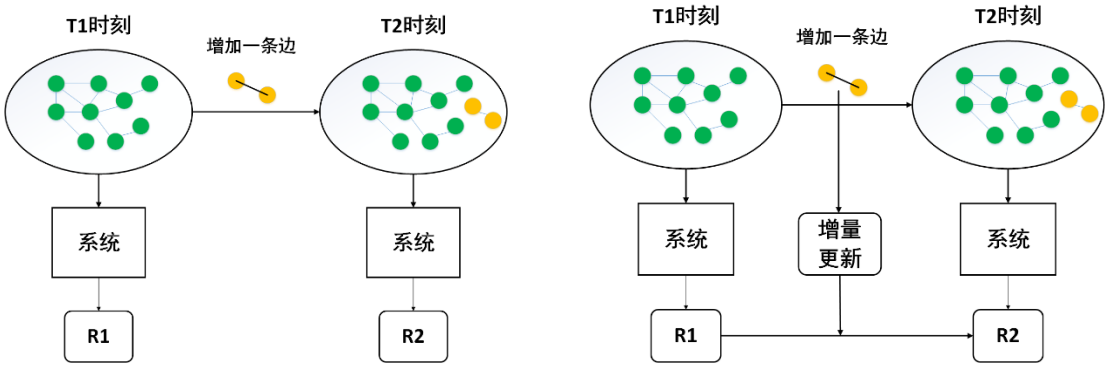
估计计算是希望通过采样或者设计精简数据结构的方式，不存或者存储少量的图数据来降低时间和空间开销，根据模型是否存储流中的数据，估算模型主要有以下两种方法：

- 采样(sampling)：该模型完全不存储流中的数据，在流经过时，对流数据进行采样和计算。该模型的内存消耗主要在采样线程上，要采 n 个样本，就要起 n 个线程对流进行采样，即每个线程只能采一个样。依赖该模型的算法的最终计算结果，取决于图结构和采样结果。例如，文献[6][7][8]采用该模型来估算图的三角形数目。
- 概要(summarization)：该模型通过将图结构转化为简单的数据结构，保存图中元素，使得消耗内存量远远小于原图。同时，结构随数据流进行更新。概要的生成方式主要有以下三种：
  - 生成树(spanner)[39][40]：该方法仅保留边的一个集合(set)，即将图结构转化为集合(set)，可以用于判断图的连通性(connectivity)和图中任意两点的距离。
  - 稀疏图(sparsifier)[41][42]：该方法仅保留边的一个权重矩阵，即将图结构转化为矩阵，可以用于估计图中每个连通分量(connected components)的权重。
  - 草图(sketch)：该方法又分为线性草图(linear sketch)[43][44]和同构草图(homomorphic sketch)[45]。线性草图仅保留点的一个向量和边的一个向量，即将图结构转化向量，因为丢失了图结构，线性草图支持的查询有限，如边权重和点的入度等；同构草图保留多个顶点矩阵，即将图结构转化为多个矩阵，保留了图结构，可以支持的查询有顶点查询，边查询，路径查询和子图查询等。

（2）准确计算

虽然估计算法能够在一定程度上节约内存和计算开销，但其估计的误差在实际的生产环境中往往变得不可控制，文献[12]指出，针对大体量的无法全部载入内存的图数据，近似算法的错误率在 95%-133%之间。因此现有的研究工作提出采用增量计算[46]的方式来得到更加准确的计算结果。

需要注意的是增量计算并不是专门针对流式图计算而设计的，学术界很早就对大数据领域中增量计算进行了研究，如 Google Percolator 系统[47]在 2010 年 6 月就上线，它以增量的形式对系统中的索引进行快速更新，这使得更新周期比原先重建索引的方式快了 100 倍左右。



（a）原始 BSP 模型　　（b）增量计算模型

图 2-10 原始计算模型与增量计算模型对比图

如图 2-10 所示，诸如 Pregel 这样的非增量计算模型，在 T1 时刻，根据当前的图数据计算得到结果 R1，在 T2 时刻，图新增一条边之后，将新增的边和原始的图数据合并构成大图数据，然后在该大图数据上重新进行计算。而如果采用增量计算模型，它在面对增量数据时，是以增量的方式在原有的计算结果 R1 上进行更新，仅计算增量数据带来的影响，这样极大程度的复用了 T1 时刻的计算结果，实时性强，计算效率更高。

2.3.2.3 KineoGraph 系统

KineoGraph[13]是一个支持增量计算的准实时流式图计算系统。它采用 VC 编程模型，以增量的方式执行更新操作。KineoGraph 的系统架构图如图 2-11 所示。

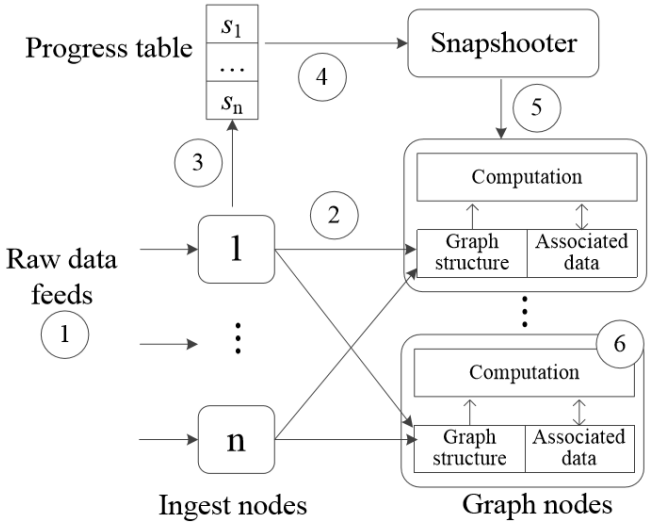


图 2-11 KineoGraph 系统架构图

各个组件的功能如下：

a. 接收节点（Ingest Nodes）：作为系统的入口，接收一系列的原始数据，并为它们创建带有唯一序列号的事务，分发给图节点；

b. 图节点（Graph Nodes）：存储图节点的信息和新增的图数据，并且完成分发过来的事务的计算，并向全局进度表汇报当前图的更新进度；
c. 全局进度表（Progress Table）：存储各个图节点的更新进度，并且向快照器提供各个图节点的更新状态；
d. 快照器（Snap Shooter）：周期性的对图节点中的进度进行数据快照操作，以增量的形式将内存里的数据输出到磁盘进行存储。

由于 KineoGraph 等计算模型都是基于增量消息进行计算，模型表达能力有限，而且更新都是串行执行，实时性有限[14]。北京大学推出了基于并发更新的分布式实时图计算模型 SpecGraph，它假设顶点状态只依赖于接收到的邻居信息，而与原来的状态无关，在该假设基础上提出了解耦合的计算模型，并通过异步执行引擎和基于推测执行的并发更新机制，进一步的提高了系统的并发度。

2.3.3 常见图系统总结

现有常见的图计算系统如表格 2-1 所示。

表格 2-1 常见图计算系统

系统类型	系统	数据划分	编程模型	计算模型
批式图计算系统 **适合离线计算**	Spark GraphX	顶点随机哈希	VC	BSP
	Flink Gelly	顶点随机哈希	VC	BSP
	Pregel	顶点随机哈希	VC	BSP
	GraphLab	顶点随机哈希	GAS	异步执行
	PowerGraph	顶点随机哈希 Edge-Cut Vertex-Cut 启发式 v-cut	GAS	混合
	Giraph	顶点随机哈希	VC	BSP
流式图计算系统 **适合在线分析**	KineoGraph	顶点随机哈希	VC	增量计算
	IncGraph	未明确表示	VC	增量计算
	SpecGraph	未明确表示	VC	增量计算

通过分析该表，可以得出以下结论：（1）针对批处理的图系统，除了 PowerGraph 这类专门处理符合 Power Law 规则的专用系统外，大部分图计算系统都是采用了顶点随机哈希（本质上是切边法）的划分原则，VC 的编程模型和同步计算泛型，而 PowerGraph 采用了启发式的切点法来划分数据，细粒度的 GAS

编程模型和异步执行引擎，因此性能表现更加出色；（2）针对流式图计算系统，一般也是采用 VC 编程模型，但在计算模型上，一般都采用了增量的方式来对图数据进行更新和计算；（3）相比较批图计算，流图计算起步较慢，更多的研究都是以算法为中心，针对某个算法来进行优化，像 Pregel、GraphLab 这种商业界和学术界都认可的批式图计算系统少之又少，因此存在很大的发展空间和研究价值。

2.4 存在问题

通过对现有图计算的关键技术分析和常见图计算系统的调研，发现现在图计算领域还存在以下问题：

（1）从**发展前景**来看，现有的图数据是海量而且动态的。批式图计算模型和系统能够有效解决海量图数据问题，但针对动态图数据，存在效率低下、资源浪费、迭代周期长和无法实时计算等问题，而流式图计算能够以增量的形式对原始数据进行更新，并且实时反馈计算结果，因此流式图计算将逐步成为处理海量动态图数据的重要手段。

（2）从**系统建设**来看，批式图计算的研究和系统建设已经非常成熟，像 Pregel 和 PowerGraph 这类离线图计算系统已经广泛得到大家的认可；而针对流式图计算的系统建设还存在很多不足，不仅现有的成型产品少，而且认可度也不高，因此流式图计算系统还存在很大的发展空间。

（3）从**研究方向**来看，现有的流式图计算大多集中在算法层面上，希望能够针对某个具体的算法（如文献[6][7][8]针对 Triangle Count），通过采样或者概要的方式来进行计算。这样针对某个具体算法进行优化的方式无法有效的推广开来，缺乏对整体的图算法特征的归纳总结，也就无法建立更为通用的计算模型。而且现有的流式图计算系统也存在各种缺陷，诸如 KineoGraph 和 IncGraph 虽然提出了增量计算模型，但其采用串行更新的方式，实时性有限；而 SpecGraph 针对此问题进行了改进，但其推出的解耦合的计算模型是建立在顶点状态只依赖于接收到的信息，而与原来的旧状态无关的假设基础上，这种假设的局限性很大，如 SSSP 算法中顶点的状态不仅与接收的消息有关，还会与顶点之前的状态有关。因此该模型的表达能力有限。

针对上述问题，本文希望通过分析现有图算法的特点，抽取图算法在流式场景下的典型特征，然后在这些特征的基础上归纳总结，构建面向流式图数据的计算模型，并且利用该模型来实现典型的流式图算法，并通过实验来验证模型和算法的正确性和实时性。

2.5 本章小结

本章首先分析了图计算中存在图数据海量和动态特性，图算法局部性差和迭代计算特点，然后从图的划分、编程模型和计算泛型三个角度阐述了现有的图计算的关键技术，接着按照批图计算和流图计算分类，对现有的常见的图计算系统进行归纳和总结，最后分析了图计算领域中存在的问题，并给出了本文的解决思路。

第三章　基于状态更新传播的流式图计算模型

在本章中，首先分析了 DD、TC、SSSP、PR 四个典型的图算法，然后详细阐述了基于状态更新传播的流式图计算模型的组件以及状态的存储和更新过程，最后列举 CC 算法说明如何在该模型上设计流式图算法。

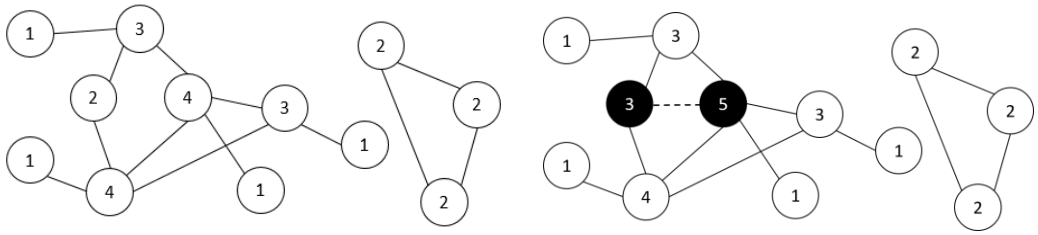
3.1 流式场景下的图算法特征分析

传统的批处理模式的图算法，在流式场景下已经不再适用。因此为解决流式场景下的图计算问题，首先需要分析典型的图算法在流式场景下所呈现的特点，再根据这些特点来设计通用计算模型。在本章节，本文选取了 DD、TC、SSSP、和 PR 四个算法来进行分析，之所以选定这四个算法，是因为这四个算法是图计算中最常见最通用的算法，而且也是其它算法的基础算法，非常具有代表性。本文约定，顶点的状态即为特定算法下顶点的值，顶点的更新函数为当有新的图数据流入系统时，顶点的状态该如何进行更新。

3.1.1　顶点度分布算法

DD 算法用于统计无向图中各个顶点的度，是图计算中最基本的算法。如图 3-1 展示了在流式场景下如何统计顶点的度。图中的圆圈表示图的顶点，圆圈之间的连线表示图的边，圆圈内部的数字表示当前时刻该顶点的度。

在 T1 时刻，系统中的各个顶点的度的分布如图 3-1(a)所示，在 T2 时刻，有一条新边（图 3-1(b)中虚线所示）流入系统。当这条边进入系统后，这条边对应的源顶点和目标顶点的度分别增加 1，而其它顶点的度保持不变。增加该条边后各个顶点的度分布如图 3-1(b)所示。



(a) T1 时刻　　(b)T2 时刻

图 3-1 无向图中各个顶点的度

由此可见，增加的这条边只影响了这条边的源顶点和目标顶点。在顶点值更

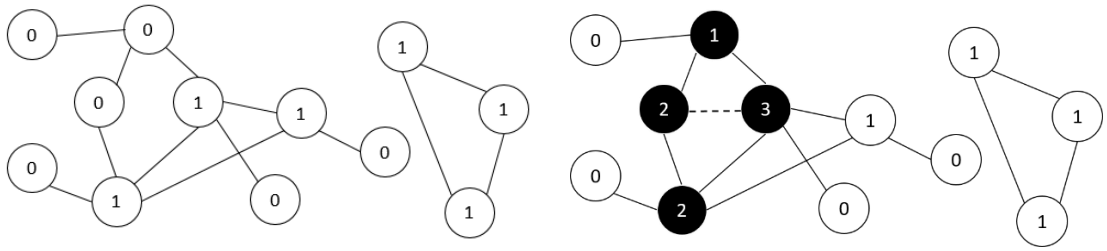
新过程中，是利用原有的顶点状态进行更新（即在原有的顶点的度的基础上进行加 1），顶点的更新顺序是无关紧要的，而且更新操作只需执行一次。

3.1.2 三角形计数算法

TC 算法是用来统计无向图中的不同三角形的数目。该算法在复杂网络分析、链接标签和推荐等多个领域中都是非常基础的重要的度量，也是一些诸如复杂网络、聚集系数等图运算中的基本方法。图 3-2 展示了在流式场景下如何统计三角形的数目，以下简称 TC 值。图中的圆圈表示图的顶点，圆圈之间的连线表示图的边，圆圈内的数字表示该顶点所拥有的三角形的数目。

在 T1 时刻，图中各个顶点的 TC 值如图 3-2(a)所示，在 T2 时刻，有一条新边（图 3-2(b)中虚线所示）流入系统，当这条边进入系统之后，与 T1 时刻相比，T2 时刻有 4 个顶点（图 3-2(b)中标黑的顶点）的 TC 值发生了变化，这条边的源顶点和目标顶点的所有公共邻接点的 TC 值增加 1，而这条边的源顶点和目标顶点的 TC 值增加的值等于这两个点的公共邻接点的数目。

由此可见，对于 TC 算法，增加一条边，不仅影响了这条边的两个顶点，还影响了这两个顶点的所有公共邻接点。在顶点值更新过程中，是利用原有的顶点状态进行更新（即在顶点原有 TC 值的基础上进行累加），顶点的更新顺序是无关紧要的，而且更新操作只需执行一次。



（a） T1 时刻　　（b）T2 时刻

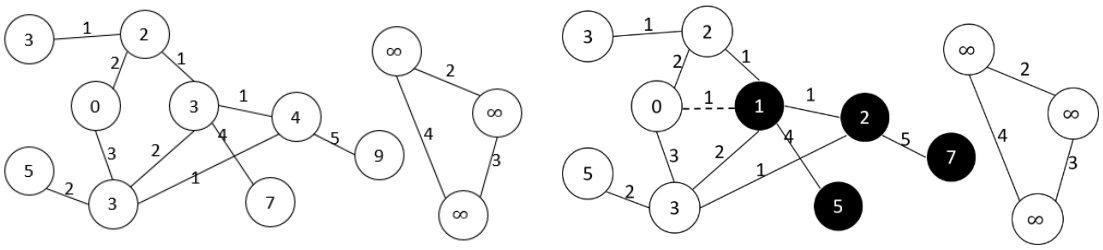
图 3-2 流式场景下的 Triangle Count 算法

3.1.3 单源点最短路径算法

SSSP 算法是解决有向图中，给定一个源点，求解这个源点到图中其它各个顶点的最短路径问题。最短路径问题是图论算法中的经典问题，也是诸如路径规划、物流规划、GPS 导航、社交网络等现实世界中许多应用的基本问题[47]。图 3-3 展示了在流式场景下如何计算各个顶点到源点的最短路径值(以下简称 SP 值)。图中边上的数字表示这两个顶点之间的距离，顶点内的数字表示当前时刻源点到该顶点的最短距离，数字为 0 的顶点为源点。

在 T1 时刻，图中各个顶点的 SP 值如图 3-3(a)所示。在 T2 时刻，有一条新边（图 3-3(b)中虚线所示）流入系统，当这条边进入系统之后，与 T1 时刻相比，T2 时刻有 4 个顶点（图 3-3(b)中标黑的顶点）的 SP 值发生了变化，新增加的这条边没有改变源点的 SP 值，但改变了目标顶点的 SP 值，接着又改变了目标顶点的后续顶点，而且这种变化是沿着某条路径传播下去的，如图中的 0->1->2->7 和图中的 0->1->5。

由此可见，对于 SSSP 算法，增加一条边，可能会影响这条边所在的顶点，甚至会以其中某个顶点为中心，将这种影响继续传播给后续顶点。在顶点 SP 值更新过程中，利用原有的顶点状态进行更新（即顶点原有 SP 值和这条边所形成的路径进行比较，取最小值），顶点的更新顺序是无关紧要的，但更新操作可能会被多次触发，如图 3-3(b)中标黑的 2 顶点可能会受标黑的 SP 值为 1 的邻接顶点的影响，也可能会受白色的 SP 值为 3 的邻接顶点的影响。



(a) T1 时刻　　(b)T2 时刻

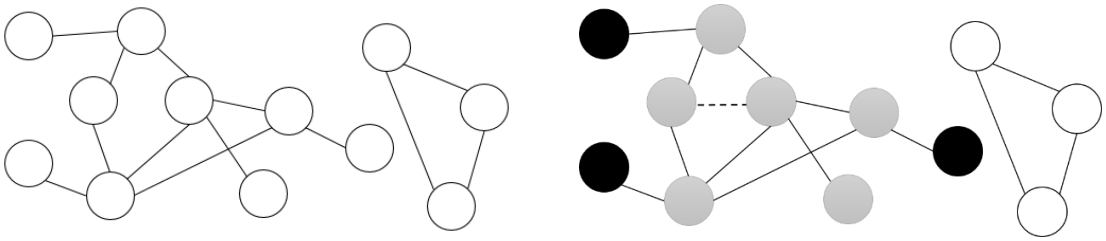
图 3-3 流式场景下的 SSSP 算法

3.1.4 PageRank 算法

PageRank[49]算法是基于网页链接来计算各个网页的重要程度算法。假设网页 A 引用网页 B，则 A 将一定的分数贡献给了 B，而网页之间是相互引用的，因此经过若干次的贡献传播之后，网页的得分会趋于稳定，该分数（即 PageRank 值，以下简称 PR 值）就是网页的重要程度的体现。图 3-4 展示了在流式场景下如何计算各个顶点的 PR 值。

在 T1 时刻，各个顶点的 PR 值如图 3-4(b)所示，在 T2 时刻，有一条新边（图 3-4(b)中虚线所示）流入系统，该边进入系统后，这条边的源顶点和目标顶点向其邻接点的贡献分数发生了变化：假设源顶点原来的出度为N，原来的 PR 值为pr_0，新增一条边后出度变为N+1，那么它对于邻接点的贡献由原来的pr_0/N变化为$pr_0/(N+1)$，进而影响了这两个顶点的所有邻接点的 PR 值，如图 3-4(b)中灰色顶点所示，随后这些灰色顶点又将这些影响继续往外传播给黑色顶点，经过若干次的迭代之后各个顶点的 PR 值保持稳定，算法运行结束。

由此可见，对于 PageRank 算法，增加一条边，这条边会影响这条边所在的连通子图内的所有顶点。在顶点 PR 值更新过程中，是利用原有的顶点状态进行更新(如新增边的源顶点和目标顶点重复利用了原始的 PR 值)，顶点的更新顺序是无关紧要的，但由于 PR 算法需要反复迭代进行计算，所以更新操作可能会被多次触发。



(b) T1 时刻　　(b)T2 时刻

图 3-4 流式场景下的 PR 算法

3.1.5 算法特征总结

下面本文将从影响范围、计算方法、计算顺序、计算特性和计算次数五个维度来分析这 4 个算法在流式场景下的特点。

（1）影响范围是指新增加的这条边可能会影响到哪些顶点的状态。例如 DD 算法的影响范围是新增边的源顶点和目标顶点；而 TC 算法的影响范围是不仅包括这两个顶点，还包括这两个顶点的所有公共邻接点。

（2）计算方法是指采用何种计算模型来进行计算。例如 DD 算法、TC 算法都采用增量计算模型来进行计算，即充分利用上次的计算结果，根据增量数据来进行增量式的计算。

（3）计算顺序是指被影响的顶点谁先参与计算对最终的计算结果是否相关。例如 DD 算法中，被影响的这两个顶点谁先参与计算都不会影响最终计算结果的正确性。

（4）计算特性是指被影响的顶点的更新函数满足哪些代数运算的性质。例如 DD 算法在计算过程中采用的是加法计算，而加法运算符合代数运算的交换律和结合律。

（5）计算次数是指这种更新函数是否会被多次触发，例如在 DD 算法中，新增加一条边只需要将这条边的源顶点和目标顶点对应的度数加 1，且这种运算只需要执行一次，而对于 PR 算法，新增加一条边时，这条边的源顶点和目标顶点的输出贡献将会发生变化，因此会首先影响它们所有的邻接点，这些邻接点在下一次的传播中会继续影响它们的邻接点，经过多次迭代计算之后各个顶点的 PR 值

会趋于稳定，在迭代计算的过程中，这条边所在的连通图内的每个顶点都可能参与多次计算。

需要说明的是，虽然这四个算法都是考虑新增一条边时的情景，但对于新增顶点和删除边等类似的图数据改变的情况，分析方法也类似，而新增边是最为常见的流式图计算的场景，所以以新增边来进行举例说明。

表格 3-1 流式场景下的图算法特征分析

	影响范围	计算方法	计算顺序	计算性质	计算次数
DD	影响新增这条边的源顶点和目标顶点	利用原始状态进行增量式计算	最终计算结果和被影响的顶点的计算顺序无关	更新函数为加法运算	被影响的顶点只参与计算一次
TC	影响新增这条边的源顶点和目标顶点，以及这两个点的公共邻接点	利用原始状态进行增量式计算	最终计算结果和被影响的顶点的计算顺序无关	更新函数为加法运算	被影响的顶点只参与计算一次
SSSP	以这条边的某个顶点为起点，沿着某条路径往其它顶点传播影响	利用原始状态进行增量式计算	最终计算结果和被影响的顶点的计算顺序无关	更新函数为 Min 运算	被影响的顶点可能会参与计算多次
PR	影响这条边的源顶点和目标顶点所在的整个连通子图内的所有顶点	利用原始状态进行增量式计算	最终计算结果和被影响的顶点的计算顺序无关	更新函数为累加运算	被影响的顶点一般会参与计算多次

通过分析这四类的典型图算法，本文发现在流式场景下这些算法的一些共性特点：

（1）计算方法满足增量计算特性：在流式场景下，本文对这些算法进行改进的首要前提是这些算法能够充分利用原始的计算结果，在原始计算结果基础上根据新增的图数据来进行增量计算。如 DD 算法，对于新增的边的源顶点和目标顶点，只是在原来顶点度的基础上各自增加 1，而其它顶点的度都保持不变，这种增量计算方式充分利用了原始的计算结果，甚至大部分顶点的度的值都不会改变，因此采用增量计算的形式大大减少了流式场景下的顶点更新代价。

（2）计算顺序满足序列一致性：在确定新增的边的影响范围后，这些被影响的顶点的状态（此处状态可以理解为顶点的值，如在 TC 算法中，顶点的 TC 值即为该顶点的状态，在模型定义中有详细的解释）该以何种顺序进行更新呢？如果被影响的顶点的更新顺序与最终的计算结果无关，本文认为这样的计算满足序列一致性原则。如上文的四个算法都符合序列一致性的计算原则，因此被影响的顶点谁先更新不会影响最终计算结果。

（3）计算性质满足代数运算的交换律和结合律：对于 DD 算法和 TC 算法，它们的更新函数是在原始的值的基础上加上一个常量，对于加法运算，显然满足交换律和结合律；对于 SSSP 算法，它的更新函数是求原始值和新值的最小值，即它的更新函数为$\min(a,b)$，而该函数满足：①$\min(a,b) = \min(b,a)$，②$\min(a,\min(b,c)) = \min(\min(a,b),c)$，即min函数也满足交换律和结合律；对于 PR 算法，每个顶点的 PR 值计算公式为：

$$PR(v) = d\sum_{v_i \in N_-(v)} \frac{PR(v_i)}{|N_+(v_i)|} + \frac{1-d}{|V|} \qquad \text{（公式 1）}$$

V为当前时刻图的所有顶点的集合，$|V|$为图的顶点数目，$v_i \in N_-(v)$为指向顶点v的所有顶点集合，$N_+(v)$为顶点v指向其它顶点的集合，d为调整因子，使得系统可以按照一定的概率跳转向图中其它任意顶点，是为了防止链接分析中链接陷阱现象的出现。对于该更新函数，考虑到公式中$(1-d)/|V|$和d均为常量，因此只分析公式中的$\sum_{v_i \in N_-(v)} \frac{PR(v_i)}{|N_+(v_i)|}$。设指向顶点$v$的顶点为$v_{k_1}, v_{k_2}, \ldots, v_{k_m}$，则原式可化简为：

$$\sum_{v_i \in N_-(v)} \frac{PR(v_i)}{|N_+(v_i)|} = \left(\frac{PR(v_{k_1})}{|N_+(v_{k_1})|} + \frac{PR(v_{k_2})}{|N_+(v_{k_1})|} + \cdots + \frac{PR(v_{k_m})}{|N_+(v_{k_m})|} \right)$$

对于m个元素的求和函数，也是满足交换律和结合律的。因此，这四个算法在流式场景下，其更新函数都满足交换律和结合律。

通过分析流式场景下图算法的典型特点，本文认为当图算法满足（1）计算方法可以采用增量计算形式；（2）计算顺序满足序列一致性原则；（3）计算函数满足代数运算的交换律和结合律 这三个特点时，本文可以实现面向流式图数据的算法（3.3.3 节会证明这个结论）。为此，本文首先建立了基于状态更新传播的流式图计算模型，并且在该模型的基础上实现了这些典型算法，下面将重点阐述该计算模型。

3.2 模型定义及 API

传统的图计算模型（例如 BSP 模型）中，图数据是静态的，本文提出的基于

状态更新传播的流式图计算模型，能够很好地解决图不断变化时的流式图计算问题。基于状态更新传播的流式图计算模型，将图在每个时刻抽象成一个对应的状态（State），将流动的图数据抽象成一系列事件流（Event Stream），事件（Event）触发了图由一个状态转换（Transform）成另一个状态。

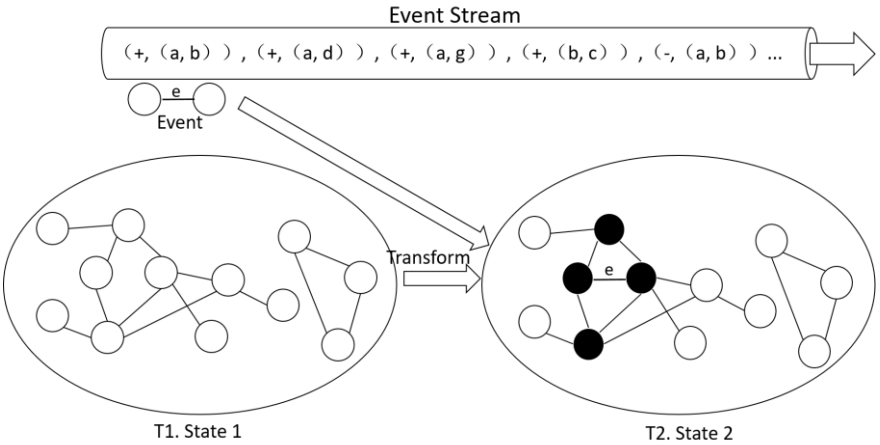


图 3-5 基于状态更新传播的流式图计算模型

如图 3-5 所示，基于状态更新传播的流式图计算模型有状态（State）、事件（Event）和转换（Transform）三个定义，下面将分别详细阐述每个组件。

（1）状态（State）

状态反应了图当前的特征信息，这些特征信息可以以顶点为单位进行体现，也可以由用户自定义的特征信息来体现，状态是由因子（Factor）组成，因子是指组成状态的基本单位，如状态可以以顶点的方式组织，那么这里的因子就是顶点。本文将状态抽象成一个接口，该接口的基本方法表见表 3-2，用户可以扩展该接口来实现更加复杂的状态信息管理。

表格 3-2 State 接口方法表

方法签名	方法作用
State GET-STATE(Factor)	获取指定因子的状态
SET-STATE(Factor, State)	设置指定因子的状态
SET-STATE(State)	设置整个图的状态
Map GET-STATE()	获取整个图的状态
SPREAD-TO-OUT-NEIGHBOR(State)	传播因子状态到邻接点

需要注意的是，状态反应了用户的关注点，虽然是根据流动的图数据而动态计算生成的，但并不等价于图数据本身，即状态不直接存储原始的图数据，而只存储用户关心的图的某些特征信息。这使得系统无需存储庞大的原始图数据，只需要存储设计精巧的状态信息即可反应图的特征信息。例如当统计图的边数时，State 可以设计为一个计数器，该计数器反应了当前时刻流入系统中的图的边数，

每次新增或者删除边时，增加或减少这个计数器的值，即可实时反应当前图的边数信息。

（2）事件（Event）

事件触发图由 T1 时刻的 State1 转换为 T2 时刻的 State2，事件是由事件值（Event Value）和事件类型（Event Type）组成。如增加一条边 e(v1,v2)这个事件中，e(v1,v2)是事件的值，增加是事件的类型。一般来说，事件的值分为两种：（顶点编号，顶点的值）和（边起点，边终点，边值）；而事件的类型分为三种：新增（ADD），删除（DELETE），更新（UPDATE）。这样可以组合出 6 种事件：新增边，删除边，更新边；新增顶点，删除顶点，更新顶点。这 6 种事件基本涵盖了所有的图变化的情形。事件的接口方法表见表 3-3。

表格 3-3 Event **接口方法表**

方法签名	**方法作用**
Value GET-VALUE(Event)	获取指定事件的值
Type GET-TYPE(Event)	获取指定事件的类型

（3）转换（Transform）

转换是由事件触发的图的更新过程，即图是如何根据相应的事件来由 State1 转换成 State2。如图 3-5 所示，在 T1 时刻图的状态为 State1，在 T2 时刻，图接收了一条边 e，这个事件会触发图状态转换函数（Transform），将图的状态转换为 State2。转换函数是动态图计算模型中的计算逻辑，详细定义了图如何根据到达的事件，从一个状态转变成另外一个状态，可以称之为状态更新的图计算模型的驱动程序，驱动图从一系列的事件流转换成一系列对应的状态流。状态的接口方法表见表 3-4。

表格 3-4 Transform **接口方法表**

方法签名	**方法作用**
TRANSFORM(State, Event)	根据事件转换状态

3.3 状态存储和更新

3.3.1 状态类型

基于状态更新传播的流式图计算模型中，一个核心问题是状态如何存储和更新。状态是从用户的视角来进行设定的，即用户关心什么数据，就可以将该数据设置为图的一个状态，这些状态可以以顶点为单位进行保存：图的状态由各个顶点的状态组成，也可以以边或者其它的方式来组织。相比较传统的顶点编程模型

或边编程模型来说，用一个高度可自定义的状态能够直接反应用户关心的结果，使得模型的表达能力更强。

根据上述 3.1 节中流式场景下的图算法特征分析，在图计算中大致分为两类状态：独立状态和关联状态。所谓独立状态，是指状态内的各个因子之间是独立的，一个因子的状态的变化不会引起其它因子的状态的变化，如 DD 算法就是属于独立状态范围，每增加一条边，这个事件只会影响增加这条边的两个顶点，不会影响到其它的顶点；所谓关联状态，是指状态内的各个因子之间相互关联，一个因子的状态的变化会影响到其它因子状态的变化，诸如 TC、SSSP、PR 算法中增加一条边，不仅会影响增加这条边的两个顶点的状态，还会影响到这两个顶点的公共邻接点，甚至整个连通子图内的所有顶点。

考虑到在计算方法、计算顺序和计算性质上这些算法特征相同，而在影响范围和计算次数上这些算法差异较大，因此可以只考虑这两个维度的变化情况，对图算法的状态进行分类，如图 3-6 所示，将问题域划分成独立状态和关联状态两种问题。

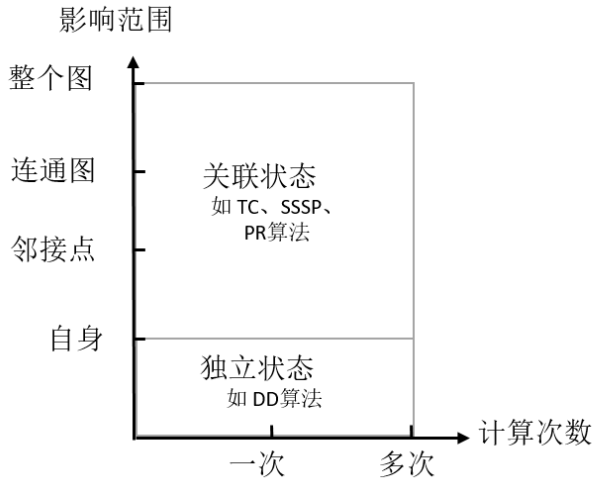


图 3-6 状态分类图

3.3.2 独立状态的更新

对于独立状态，因为状态内的各个因子之间不会相互影响，因此独立状态可以并发地更新。即可以按照状态的组织形式，将图的状态分布式地存储在多个计算节点上，而且每个计算节点上的状态都可以同时进行更新，并向用户实时反馈更新结果。这样充分利用了分布式的特点，提高存储和计算效率。

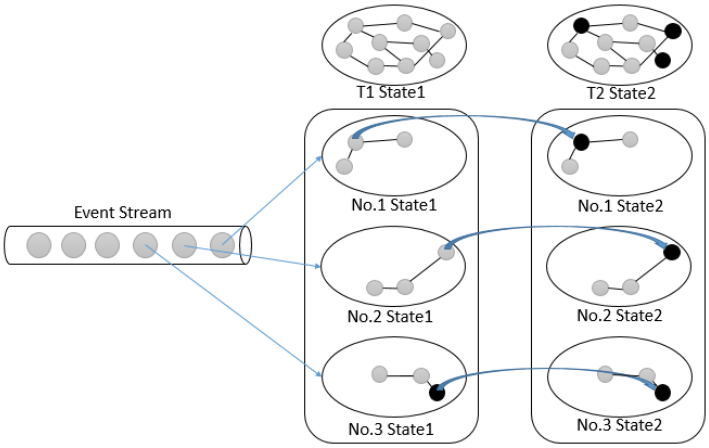


图 3-7 独立状态的存储和更新过程图

图 3-7 演示了独立状态的并行更新的过程。系统接收到事件流（Event Stream）之后，将事件流按照某种分片规则（即特定的图的划分算法），分发到不同的计算节点上（如图所示的 No.1、No.2、No.3 这 3 个计算节点），然后分别在各个计算节点上独立进行状态更新（如图所示对应计算节点 No.1、No2、No3 的状态从 State1 转换到了 State2），这些计算节点都更新完毕之后，图的状态就由 T1 时刻 State1，更新成了 T2 时刻的 State2，注意到更新过程充分利用了分布式的优势，多个事件可以分配到不同的计算节点上同时进行更新，提高更新效率。

3.3.3 关联状态的更新

在关联状态中，一个因子的状态的变化会影响到其它因子状态的变化，因此，多个事件触发的更新可能会影响同一个因子，引起更新冲突问题。解决更新冲突的方法有很多，最为简单的方式是将多个事件的更新串行化，即对两个事件 A 和 B，事件 A 先触发更新，A 更新完毕后事件 B 再触发更新，在事件 A 触发更新期间，其它任何事件都不得触发更新，以免引起更新冲突问题。诸如 IncGraph 就是采用这样的更新模型，这使得即使不会发生更新冲突的两个事件也不能同时更新，无法充分利用多机并行的优势。因此本文提出了两种解决更新冲突的方法：基于分区的并行更新策略和基于细粒度分布式锁的并行更行策略。

基于分区的并行更新策略是将原来的图划分成若干个子图，使得子图内部的顶点联系比较紧密，子图之间的顶点几乎没有边相连或者联系较少。这样可以假设子图内顶点更新的影响范围只限于子图内部，不会传播到其它子图中顶点。因此子图与子图之间的更新可以同时进行，而子图内部的更新则需要串行进行，这样在一定程度上能够提高更新的并行度。

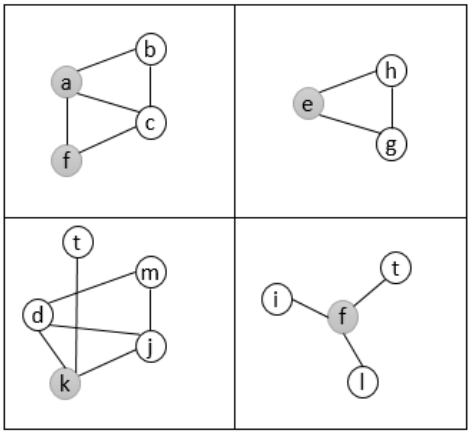


图 3-8 基于分区的并行更新策略

图 3-8 展示了基于分区的并行更新策略，按照连通性将原来的图分成四个连通子图，在每个连通子图内部的更新是串行的，如在左上角子图中 a 顶点和 f 顶点的更新需串行进行，而连通子图之间的顶点的更新是并行的，如 a 顶点、e 顶点、k 顶点和 f 顶点的更新可以同时进行。如果分区策略划分的好，可以充分利用分布式的优势，实现多个分区并行更新。

基于分区的并行更新策略需要谨慎地选择子图划分算法，该分区算法要能够很好的将原来的大图切分成若干个子图，保证子图之间顶点的联系是松散的，子图内部的顶点之间的联系是紧密耦合的。关于动态图划分的问题，现有的研究工作也很多，Ioanna Filippidou 和 Yannis Kotidis[27]以及 Stanton 和 Kliot[28]都提出了很多启发式的算法，本文在相关工作中已经介绍，而本文的工作重心不是比较这些图划分算法的优劣，而是希望借鉴这些现有的图划分算法来提高系统的并行更新能力。

基于分区的并行更新策略不可避免得会出现多个顶点的更新会集中在一个子图上的情况，这种情况会严重影响系统整体的并行度。基于分区的并行更新策略本质上是一个范围锁，锁住一定范围内的所有顶点数据。这样粗粒度的锁会大大影响系统的并发性，因此本文又提出了基于细粒度分布式锁的并行更新策略，即每次只需要锁住组成状态的单个因子本身，而不需要锁住范围内的所有顶点。如图 3-9 所示，该锁以因子为单位，每次访问因子时，首先检测该因子是否是被占用状态，如果没有被占用，则首先将该因子设置为已占用，然后更新因子的状态，更新完毕之后，再将因子设置为空闲状态，同时满足一定阈值时，将已被更新的因子的状态传播给其它顶点。

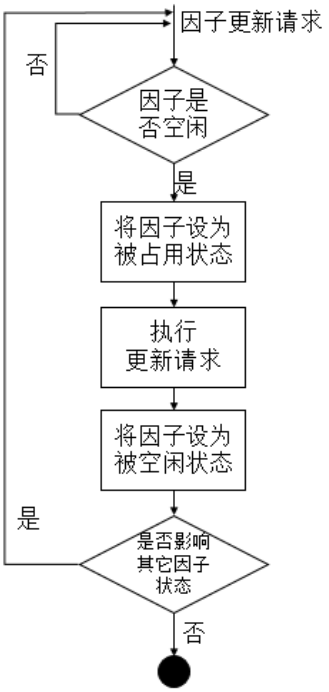
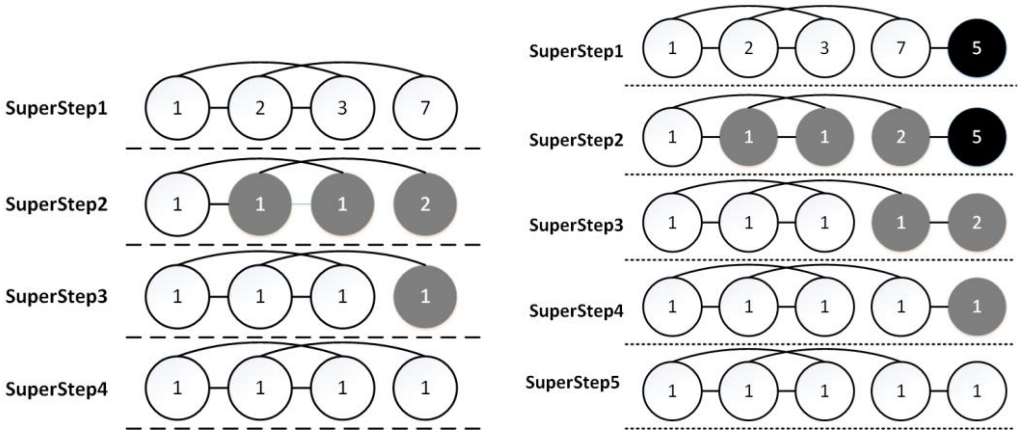


图 3-9 基于细粒度分布式锁的因子更新过程图

基于细粒度分布式锁的更新方式是增量式的变化传播更新。所谓增量式更新即充分利用原有的计算结果，在原始结果上根据增量数据带来的影响进行局部范围的更新，在 3.1 节算法分析中，本文详细阐述典型图算法在流式场景下的特征，其采用的计算模型即为增量计算模型，也就是这里所述的增量式更新；所谓变化传播更新，是指当顶点的状态发生变化时，该顶点将其变化的状态传播给其相关的顶点（例如该顶点的所有邻接点），而这些被影响的邻接点再根据自身的状态变化情况，决定是否要继续传播这种影响。

下面将举例说明这种策略相对传统的 BSP 模型的优势。图 3-10(a)是使用传统的 BSP 模型来将图中各个顶点的值设置为图中最小标号的值的运算过程。图的初始状态如 SuperStep1 所示，经过 4 次迭代之后，将图中各个顶点的值都设置为了最小标号 1。当在原图中增加一个顶点时，BSP 模型的运行过程如图 3-10(b)所示：新增边(7,5)之后，BSP 模型将新增的边和原图构成一个新的图，并且在新的图上重新进行计算，经过 7 次迭代之后，完成将图中各个顶点的值都设置为最小标号 1 的过程。由此可见，传统的 BSP 模型在处理增量图数据时，没有充分考虑上次的计算结果，进行了大量的重复计算和通信过程。而采用了增量式的变化传播方式对图进行更新时，是在上次的计算结果上进行增量式的更新，而且在更新过程中，以变化传播的方式来传播增量数据所带来的影响，从而避免了全图所有顶点都参与重算的过程，减少了计算和通信开销。如图 3-11 所示，仅需要 3 次迭代，而且将增量数据的影响控制在了黑框范围内，完成了整个图的计算过程，

影响范围更小，收敛速度更快。



（a）原始数据　　（b）增量数据

图 3-10 传统 BSP 模型计算图最小标号过程

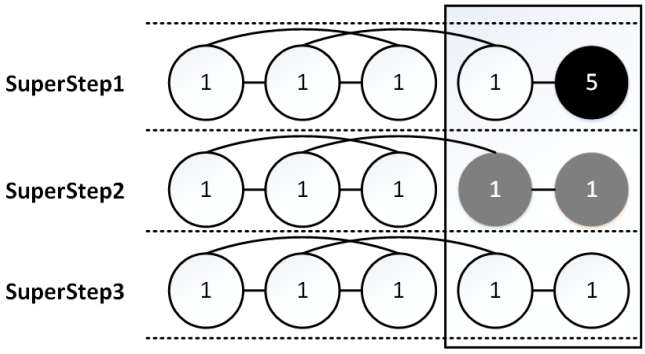


图 3-11 增量式的变化传播方式计算图最小标号过程

通过上述分析，这种基于细粒度分布式锁的更新策略，相比较传统的 BSP 模型有如下优势：

- 传统的 BSP 模型是将整个图的迭代计算过程分解为若干个超步，超步内部的顶点之间并行计算，超步之间进行同步。这使得在每个超步内，计算最慢的节点拖慢整个超步的计算速度，因此会出现短板效应，而本文的基于细粒度分布式锁的并行更新策略有效弥补了这个不足，因子（这里的因子等价于 BSP 模型中的顶点）与因子之间的更新都是并行的，只有属于一个因子的多个更新请求才会被串行执行，这样真正实现了多个因子的并行更新策略，而且没有显式的同步过程，消除了短板效应。
- 采用增量式的变化传播更新，采用增量计算的方式，能够有效减少整体迭代所需的次数，收敛更快；采用变化传播的方式，能够有效控制增量数据所来带的影响范围，减少参与计算的顶点的数目，从而减少通信和计算开销。

那么这种基于细粒度分布式锁的更新策略能否保证算法的正确性呢？下面本文将证明满足在 3.1 节总结的三个特点的条件下，在采用基于细粒度分布式锁更新策略时，能够保证算法的最终正确性。

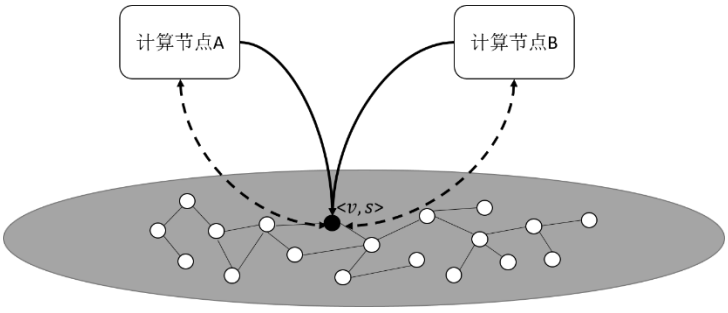


图 3-12 计算节点同时更新状态图

如图 3-12 所示，假设计算节点 A 和 B 同时请求更新顶点v的状态，而顶点v之前的状态为s，图中虚线是请求并获取顶点v的原始状态，实线是修改顶点v的状态，并同步到存储介质中。如果不控制顶点的并发更新，则可能出现 A 读取v的状态之后正准备更新时，B 也读取了v的状态，那么此时 B 读取的是 A 未完成更新的v的旧状态s，出现了脏读的情况，程序最终的计算结果是错误的。如果采用基于细粒度分布式锁的并发更新策略，则在 A 更新顶点v时，B 是无法访问顶点v的，只有等到 A 将更新后的状态s'写入到v之后，B 才能够访问并修改顶点v的状态。当状态的更新函数满足（1）计算方法满足增量计算特性；（2）计算顺序满足序列一致性；（3）计算性质满足代数运算的交换律和结合律 这三条性质时，可以将状态函数抽象为

$$\text{Transform}(v) \rightarrow state_{new} = f(state_{old}, event) \qquad \text{（公式 2）}$$

$state_{old}$为顶点v的原始状态，$event$为接收的事件，f为更新函数，$state_{new}$为事件$event$驱动顶点v更新后的状态。假设有若干个事件$event_1, event_2, \dots, event_k$请求对顶点$v$更新，由于多个计算节点同时进行更新，所以无法控制事件的更新顺序，但在使用基于细粒度分布式锁的更新策略之后，可以保证针对顶点v的更新是串行的，因此顶点v的最终状态为：

$$state_{new} = f(\dots f\big(f\big(state_{old}, event_{t_1}\big), event_{t_2}\big), event_{t_k}) \qquad \text{（公式 3）}$$

$event_{t_1}, event_{t_2}, \dots, event_{t_k}$为系统执行时事件真正更新的顺序，当计算方法满足特性（2）和（3）时，我们知道无论$event$如何排列，最终的计算结果都是一样的，因此在分布式更新环境下，算法的最终计算结果与事件的更新顺序无关，即分布式环境下，算法依旧能够计算得到正确结果。大多数算法（如本文中的 DD、TC、SSSP 等算法）都满足这个条件，因此模型的表达能力不会受到太大影响。此外，对于并行环境下的超大规模图计算问题，其实更新冲突的概率非常小，不妨假设需要计算的图有$|V|$个顶点，则两个计算节点同时更新同一个顶点状态的概率为$\frac{1}{|V|}$，如果集群中有 N 个计算节点同时更新，则至少有两个计算节点发生更

新冲突的概率为（1-$\frac{A_{|V|}^{N}}{|V|^{N}}$）。本文在 6.2.3 实验中也验证了这种冲突的概率不会超过 3%，因此采用基于细粒度分布式锁的并行更新策略可以有效提高并行更新的效率，这也是本文所采用的更新策略。

3.4 模型应用举例

基于状态更新传播的流式图计算模型从用户的角度出发来定义状态，只保存用户关心的数据，相比较传统的基于顶点的编程模型来说表达能力更强。在此本文选取了连通子图（Connected Components， CC）算法来说明如何在该模型上如何进行算法设计。

如果一个图中，每对顶点都有路径相连，则称其为连通图。如果图的子图中任意两个顶点都是可达的，则这个子图称之为图的连通分支。连通分支反应了一个大图中子图的聚集情况，可以根据连通分支将原来的大图分解成若干个连通分支，算法独立并行的在连通分支上进行。连通分支在好友推荐、循环引用判断等诸多问题上被广泛使用。

下面将介绍在不断增加边的情况下，如何针对无向图做连通分量的计算。由前文所知，基于状态更新传播的流式图计算模型有三个概念：State，Event，Transform，下面将详细介绍如何定义这三个基本组件：

（1）State：当前图的所有的连通分支，$State = \{s_1, s_2, \dots, s_n\}, n = |V|$，State = $\{\mathrm{s}_1, \mathrm{s}_2, \dots, \mathrm{s}_\mathrm{n}\}$，其中$s_k$表示第$k$个连通分支，$s_k = \left(v_{k_{min}}, \{v_{k_1}, v_{k_2}, \dots, v_{k_r}\}\right)$，集合$\{v_{k_1}, v_{k_2}, \dots, v_{k_r}\}$表示由这些顶点构成了一个连通分支，$v_{k_{min}}$是这些顶点中标号最小的点。

（2）Event：图的 Event 为图中新增了一条边，那么这些 Event 构成的序列就形成事件流，即$Event\ Stream =_1, z_2, \dots, z_m$，其中$z_k = (e_k, ADD)$，表示新增边$e_k$ $\mathrm{z_k} = (\mathrm{e_k}, \mathrm{add})$；

（3）Transform：在系统启动时，还没有任何数据流入系统中，此时状态集合为空，当图的边流数据慢慢进入系统时，State 的更新过程见算法 1。

在算法 1 中，s_1表示顶点v_1所在的连通分支，s_2表示顶点v_2所在的连通分支，新增的这条边$e = (v_1, v_2)$使得这两个顶点所在的连通分支合并，构成一个大的连通分支，算法中用户自定义的 UNION-STATE()函数完成两个连通分支的合并。需要注意的是，算法 1 中省略了针对顶点进行加锁和释放锁的过程，根据本文提出的基于细粒度分布式锁的更新策略，在获取某个顶点的状态之前，需要先检测或者锁住这个顶点的状态，因此在算法的开始和结束位置有锁的获取和释放相关

代码，特此说明，以下算法类推。

算法 1 Dynamic Connected Components

```
1  for each z ∈ EventStream
2    do (v1,v2) ← GET-VALUE(z)
3      s1← GET-STATE(v1)
4      s2← GET-STATE(v2)
5      if s1≠ ∅ and s2≠ ∅
6        then UNION-STATE(s1,s2)
7      elseif s1≠ ∅ and s2= ∅
8        then UNION-STATE(s1,v2)
9      elseif s1= ∅ and s2≠ ∅
10       then UNION-STATE(s2,v1)
11     else
12       s ← (min(v1, v2),{v1, v2})
13     ADD-STATE(s)
```

3.5 本章小结

本章首先对流式场景下的 DD、TC、SSSP 和 PR 算法进行了特征分析，并且总结出满足（1）计算方法可以采用增量计算形式；（2）计算顺序满足序列一致性原则；（3）计算函数满足代数运算的交换律和结合律 这三个特点的图算法，可以改成面向流式图数据的算法；在分析完后，本文介绍了针对这些算法而建立的通用的基于状态更新传播的流式图计算模型，并且给出了模型的状态、事件和转换这三个基本组件的定义和使用规则，最后以 CC 算法举例说明如何使用该模型进行算法设计。

第四章　流式图算法设计

在本章节，使用第三章节建立的基于状态更新传播的流式图计算模型提供的组件来对 DD、TC、SSSP 和 PR 的算法进行重新设计，以适应流式图数据场景，并以这些算法为例来指导读者如何运用该模型来进行流式图算法的设计。

4.1 顶点度分布算法

DD 算法是用来统计无向图中各个顶点的度。因此图中各个顶点的度值组成了整体图的状态。因此本文定义：

（1） State：当前图中各个顶点的度值，即$State = \{s_1, s_2, \dots, s_n\}, n = |V|$，其中$s_k = (v_k, d_k)$，表示顶点$v_k$的度为$d_k$；

（2） Event：图的 Event 为图到达一条边相关的事件，那么 Event 构成的序列就形成事件流，即$Event\ Stream = z_1, z_2, \dots, z_m$ ，其中$z_k = (e_k, TYPE)$，$TYPE \in \{ADD, UPDATE, DELETE\}$，这三种状态对应的事件分别为增加一条边，更新一条边和删除一条边；

（3） Transform：在系统启动时，还没有任何数据流入系统中，此时状态集合为空，当图的边流数据慢慢进入系统时，State 的更新过程如算法 2 所示：

算法 2 Dynamic Degree Distribution

```
1   for each z ∈ Event Stream
2       do  (v1, v2) ← GET-VALUE(z)
3           type ← GET-TYPE(z)
4           d1← GET-STATE(v1)
5           d2← GET-STATE(v2)
6           if  type = ADD
7               then SET-STATE(v1, d1+1)
8                    SET-STATE(v2, d2+1)
9           elseif  type = DELETE
10              then SET-STATE(v1, d1-1)
11                   SET-STATE(v2, d2-1)
```

4.2 三角形计数算法

TC 算法是用来统计有向/无向图中的不同三角形的数目。该算法在复杂网络分析、链接标签和推荐等多个领域中都是非常基础重要的度量，也是一些诸如复

杂网络、聚集系数等图运算中的基本方法。

由前文所知，基于状态更新传播的流式图计算模型，有 State，Event，Transform 三个重要的概念。针对 Triangle Count 算法，这三个组件的定义如下：

（1）State：图的 State 由每个顶点对应的邻接点的信息组成，$State = \{s_1, s_2, \dots, s_n\}, n = |V|$，其中$s_k = (v_k, N_k, t_k)$，表示顶点$v_k$的邻接点的集合为$N_k$，顶点$v_k$构成的三角形的数目为$t_k$；

（2）Event：图的 Event 为图到达一条边相关的事件，那么 Event 构成的序列就形成事件流，即$Event\ Stream = z_1, z_2, \dots, z_m$，其中$z_k = (e_k, TYPE)$，$TYPE \in \{ADD, UPDATE, DELETE\}$，这三种状态对应的事件分别为增加一条边，更新一条边和删除一条边；

（3）Transform：图的状态在事件流的驱动下的转换函数，如算法 3 所示：

算法 3 Dynamic Triangle Count

```
1   for each z ∈ Event Stream
2     do N1← ∅
3        N2← ∅
4        (v1, v2) ← GET-VALUE(z)
5        s1← GET-STATE(v1)
6        s2←GET-STATE(v2)
7        if s1≠ ∅
8          then N1← GET-NEIGHBOR(s1) ∪ {v2}
9        else N1← {v2}
10       if s2≠ ∅
11         then N2← GET-NEIGHBOR(s2) ∪ {v1}
12       else N2← {v1}
13       cross ←N1∩N2
14       type ← GET-TYPE(z)
16       for each v ∈ cross
17         do sv ← GET-STATE(v)
18            t ← GET-TRIANGLE(sv)
19            N ← GET-NEIGHBOR(sv)
20            if type = ADD then sv←(v,N,t+1)
21            elseif type = DELETE then sv←(v,N,t-1)
22            SET-STATE(v, sv)
23       t1← GET-TRIANGLE(s1)
24       t2← GET-TRIANGLE(s2)
25       if type = ADD
26         then s1← (v1, N1, t1+|cross|)
```

27	s_2← (v_2, N_2, t_2 +|cross|)
28	***elseif*** type = DELETE
29	s_1←(v_1, N_1, t_1-|cross|)
30	s_2←(v_2, N_2, t_2-|cross|)
31	SET-STATE(v_1, s_1)
32	SET-STATE(v_2, s_2)

算法 3 演示了动态的 TC 算法流程。事件 z 作为算法的驱动，驱使图由当前的状态转换为下一时刻的状态。程序第 2 行和第 3 行初始化一个集合，用来存储顶点v_1和v_2的邻接点；第 4 行从事件 z 中取出新增的边的源顶点和目标顶点；第 5 行和第 6 行获取这两个顶点的状态；第 7-12 行用来更新顶点v_1和v_2的邻接点；第 13 行用来求解这两个集合的交集，即v_1和v_2的公共邻接点，这些公共邻接点将会构成新的三角形；第 16-22 行根据事件的类型来更新这些公共邻接点的状态—如果事件类型为 ADD，则将这些公共邻接点的 TC 值增加 1，如果事件类型为 DELETE，则将这些公共邻接点的 TC 值减少 1；第 23-32 行用来更新顶点v_1和v_2的状态—如果事件类型为 ADD，则将其 TC 值增加|cross|，|cross|为公共邻接点的数目，如果事件类型为 DELETE，则将其 TC 值减少|cross|。

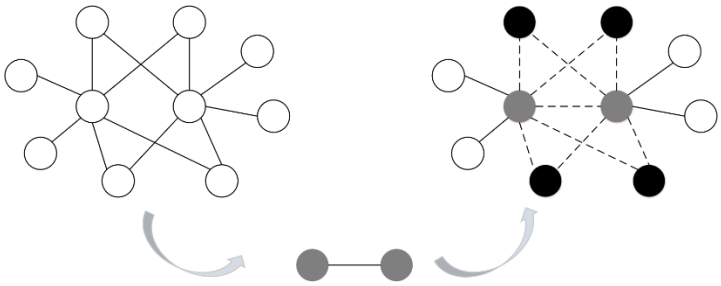


图 4-1 Dynamic Triangle Count

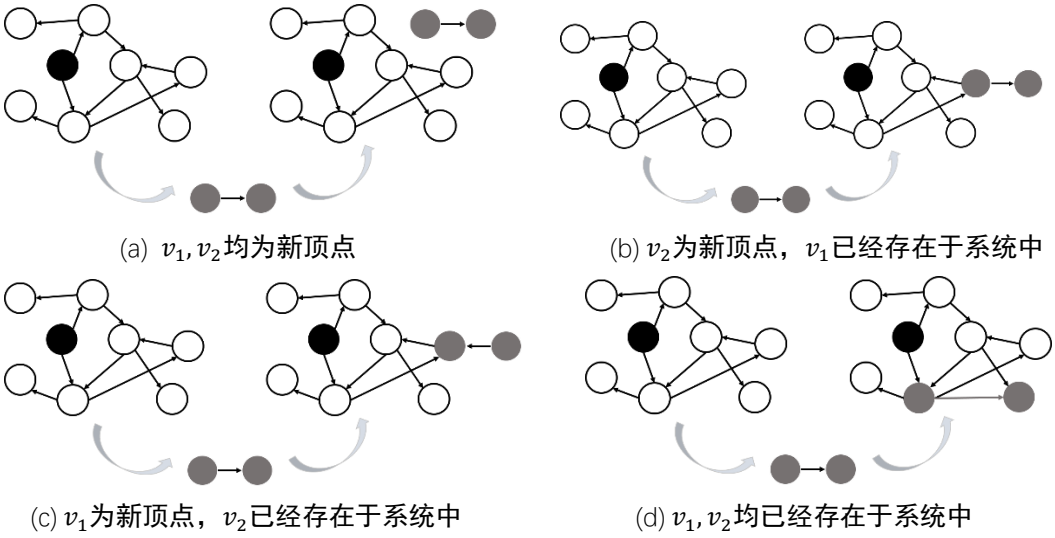
如图 4-1 所示，左边白色的图为当前时刻图的状态，在增加一条边之后，即图中灰色顶点及其连线，这会使得整个图新增了 4 个三角形（图 4-1 右图中虚线组成的三角形所示），这 4 个三角形即为新增边的两个顶点的公共邻接点（图 4-1 右图中黑色的点）所形成的三角形。因此公共邻接点的三角形数目各自增加 1，而新增边的两个顶点的三角形数目增加|cross|。

4.3 单源点最短路径算法

SSSP 算法是解决有向图中，给定一个源点，求解这个源点到图中其它各个顶点的最短路径问题。最短路径问题是图论算法中的经典问题，也是诸如路径规划、物流规划、GPS 导航、社交网络等现实世界中许多应用的基本问题。

假设系统中新增的边$e = (v_1, v_2)$，边的方向为v_1指向v_2。则一共有如图 4-2

所示的(a) v_1, v_2均为新顶点；(b) v_2为新顶点，v_1已经存在于系统中；(c) v_1为新顶点，v_2已经存在于系统中；(d) v_1, v_2均已经存在于系统中 这 4 种情况，而前面的 3 种情况只可能是增加边的事件触发（因为不可能删除原图中根本就不存在的顶点），而第 4 种情况可能是增加、更新和删除这三种类型的事件。图 4-2 中，黑色顶点是源点，白色的顶点是图中已经存在的顶点。在每个子图中，左边部分是原图，中间灰色的顶点和连线是新增的边（新增边的两个顶点编号为v_1, v_2），右图为新增边之后的新图。



(a) v_1, v_2均为新顶点　(b) v_2为新顶点，v_1已经存在于系统中

(c) v_1为新顶点，v_2已经存在于系统中　(d) v_1, v_2均已经存在于系统中

图 4-2 Dynamic Single Source Shortest Path

针对以上 4 种情况，有如下分析：

（a）v_1, v_2均为新顶点

如图 4-2(a)所示，v_1, v_2为原图中不存在的新增加的顶点，对于这两个新增加的顶点，原图中的任何顶点都无法到达这两个顶点，因此这两个顶点的 SP 值为无穷大。

（b）v_2为新顶点，v_1已经存在于系统中

如图 4-2(b)所示，v_1为原图中已经存在的顶点，v_2为原图中不存在的新增加的顶点，且有v_1指向v_2，此时，因为指向v_1的顶点集合没有变化，所以v_1的 SP 值不会发生改变；而又有v_1指向v_2，因此v_2可由v_1到达，所以v_2的 SP 值更新为$SP_{v_1} + dis_e$，其中SP_{v_1}为v_1的 SP 值，dis_e为边e的权重。

（c）v_1为新顶点，v_2已经存在于系统中

如图 4-2(c)所示，v_2为原图中已经存在的顶点，v_1为原图中不存在的新增加的顶点，且有v_1指向v_2。此时，因为是v_1指向v_2，而v_1又是新增加的顶点，那么没有任何顶点指向v_1，即v_1是不可达的，则更新v_1的 SP 值为无穷大；而指向v_2的顶点，相比较原图的情况只增加了v_1，又因为v_1是 SP 值是无穷大，所以v_2的 SP 值不会发生变化。

（d）v_1, v_2均已经存在于系统中

① ADD 和 UPDATE 事件

如图 4-2(d)所示，v_1, v_2均为原图中已经存在的顶点。因为是v_1指向v_2顶点，所以指向v_1顶点的集合没有改变，因此v_1的 SP 值也不会发生改变；而指向v_2的顶点集合中增加了v_1，这使得可能存在一条更短的路径从v_1指向v_2，因此，

$$SP_{v_2} = \min(SP_{v_1} + dis_e, SP_{v_2})$$

即取v_2的原来的 SP 值和从v_1过来到达v_2的值的最小值。如果v_2的 SP 值变小，则v_2的后续顶点的 SP 值可能因为v_2的变小而变小，因此当v_2的 SP 值变小时，需要将这种变化传播给v_2所有指向的邻接点，同时，这些邻接点又可能继续将这种影响传播出去；而当v_2的值没有发生改变时，说明从v_1过来的路径不是最短路径，v_2的值不受影响，其后续顶点的值也不会发生变化。当所有顶点的值不再发生变化时，图的状态更新完毕，算法运行结束。

②DELETE 事件

从左往右看图 4-2(d)，是新增边的过程；从右往左看图 4-2(d)，则是删除边的过程。当删除原图中的边时，首先要考虑删除的这条边是否会影响v_2的 SP 值。在原图中，当$SP_{v_1} + dis_e > SP_{v_2}$时，说明$v_2$的最短路径并不是从$v_1$过来的，即删除$(v_1, v_2)$这条边并不会影响$v_2$的 SP 值，而且也不会影响$v_1$的 SP 值；当$SP_{v_1} + dis_e = SP_{v_2}$时，并不是说$v_2$的最短路径一定是从$v_1$过来，因为还有可能存在其它的顶点到达$v_2$的路径和$SP_{v_2}$相同，因此需要判断$v_1$是否是到达$v_2$的唯一的最短路径，如果是那么此时删掉$v_1$一定会引起$v_2$的 SP 值发生改变，则更新$v_2$的 SP 值，并且将这种改变传播给邻接点，否则删除这条边不会影响任何顶点的 SP 值。

在分析完流式场景下的算法细节之后，下面本文将套用基于状态更新传播的流式图计算模型来对 SSSP 算法进行设计。算法的三个组件定义如下：

（1）State：图的 State 由每个顶点对应的邻接点的信息组成，$State = \{s_1, s_2, \dots, s_n\}, n = |V|$，其中$s_k = (v_k, sp_k)$，表示顶点$v_k$到源点的最短路径为$sp_k$；

（2）Event：图的 Event 为图到达一条边相关的事件，那么 Event 构成的序列就形成事件流，即$Event\ Stream = z_1, z_2, \dots, z_m$，其中$z_k = (e_k, TYPE)$，$TYPE \in \{ADD, UPDATE, DELETE\}$，这三种状态对应的事件分别为增加一条边，更新一条边和删除一条边；

（3）Transform：图的状态在事件流的驱动下的转换函数，如算法 4 所示：

算法 4 Dynamic Single Source Shortest Path

```
1    for each z ∈ Event Stream
2        do (v1,v2,dis_e) ← GET-VALUE(z)
```

```
3       type ← GET-TYPE(z)
4       s1← GET-STATE(v1)
5       s2←GET-STATE(v2)
6       sp1← GET-STATE(s1)
7       sp2←GET-STATE(s2)
8       if s1= ∅ and s2= ∅
9           then SET-STATE(v1,+∞)
10                SET-STATE(v2,+∞)
11      elseif s1≠ ∅ and s2= ∅
12          then SET-STATE(v2, sp1 + dise)
13      elseif s1=∅ and s2≠ ∅
14          then SET-STATE(v1,+∞)
15      else
16          candidate = sp1 + dise
17          if  type = ADD or type = UPDATE
18              if candidate < sp2
19                  then SET-VALUE(v2,candidate)
20                       SPREAD-TO-OUT-NEIGHBOR(v2, candidate)
21          elseif  type = DELETE
22              if candidate = sp2 and ONLY-SHORTEST-PATH(v1,v2)
23                  then candidate = FIND-SHORTEST-PATH(v2, v1)
24                       SPREAD-TO-OUT-NEIGHBOR(v2, candidate)
```

算法 2-7 行是获取v_1，v_2顶点（v_1为源顶点，v_2为目标顶点）对应的状态，8-10 行对应图 4-2(a)情况，即新增的两个顶点都是最新的顶点，则其 SP 值更新为无穷大；11-12 行对应图 4-2(b)情况，源顶点是原图中已经存在的顶点，目标顶点是新增顶点，则更新目标顶点的 SP 值为源顶点的 SP 值加上边的权重；13-14 行对应图 4-2(c)情况，源顶点是新增的顶点，目标顶点是原图中已经存在的顶点，则目标顶点的 SP 值保持不变，同时更新源顶点的 SP 值为无穷大；15-24 行对应图 4-2(d)情况，而这种情况根据事件类型的不同，又可以分为新增&更新，删除这两类情况，对于新增和更新事件，需要考虑这条边是否能够使得目标顶点的 SP 值变小，如果使得目标顶点的 SP 值变小，则将这种影响继续传播给目标顶点的邻接点，否则不做任何操作；算法中的 SPREAD-TO-OUT-NEIGHBOR()函数即为影响传播函数，将这个影响继续传播给该顶点的所有出去的邻接点；而对于删除事件，首先需要考虑原图中源顶点v_1是到达目标顶点v_2的唯一最短路径顶点，只有满足这个条件，删除这条边才会影响目标顶点的 SP 值，如果删除的这条边恰好是这样的关键路径，则重新计算v_2的 SP 值，同时将这种变化通过

SPREAD-TO-OUT-NEIGHBOR()函数传播出去；算法中的 ONLY-SHORTEST-PATH(v_1, v_2)即为判断v_1是否是到v_2的唯一最短路径，FIND-SHORTEST-PATH(v_2, v_1)计算除去v_1顶点之后，到达v_2顶点的最短路径的值。

4.4 PageRank 算法

PR 算法是基于网页链接来计算各个网页的重要程度算法。假设网页 A 引用网页 B，则 A 就将一定的分数贡献给了 B，而网页之间是相互引用的，因此经过若干次的迭代之后，网页的得分会趋于稳定，该分数就是网页的重要程度的体现。

类似于 SSSP 算法的流式场景分析，PR 算法也会遇到如图 4-2 所示的 4 种情况。针对这 4 类的处理过程如下：

(a) v_1, v_2均为新顶点

如图 4-2(a)所示，v_1, v_2为原图中不存在的新增加的顶点，对于这两个新增加的顶点，初始化其 PR 值为 1，又因为这两个顶点不与图中的任何顶点建立联系，根据 PR 得分计算公式(见公式 1)：

源顶点v_1的 PR 值：

$$PR(v_1) = \frac{1-d}{|V|+2}$$

目标顶点v_2的 PR 值：

$$PR(v_2) = \frac{1-d}{|V|+2} + d * 1$$

其中，$|V|$是原图中的顶点数目。

(b) v_2为新顶点，v_1已经存在于系统中

如图 4-2(b)所示，v_1为原图中已经存在的顶点，v_2为原图中不存在的新增加的顶点，且有v_1指向v_2。此时，因为v_2是新顶点，所以首先将v_2的 PR 值初始化为 1，而v_1指向其它顶点的集合增加了顶点v_2，因此v_1对其所有指向的邻接点的贡献值将发生变化，这种变化将会以v_1为中心逐步扩散开来。

(c) v_1为新顶点，v_2已经存在于系统中

如图 4-2(c)所示，v_2为原图中已经存在的顶点，v_1为原图中不存在的新增加的顶点，且有v_1指向v_2。此时，因为v_1是新顶点，所以首先将v_1的 PR 值初始化为 1，又因为没有顶点指向v_1，所以顶点v_1的 PR 值：

$$PR(v_1) = \frac{1-d}{|V|+1}$$

而对于v_2来说新增了v_1顶点指向它，所以它的 PR 值在原来的基础上增加v_1顶点的贡献值，即

$$PR(v_2) = PR(v_2) + d * 1$$

此时v_2顶点的 PR 值发生了变化，则以v_2为中心将这种变化传播出去。

(d) v_1, v_2均已经存在于系统中

① ADD 事件

如图 4-2(d)所示，v_1, v_2均为原图中已经存在的顶点，v_1指向v_2。对于v_1顶点，指向v_1的顶点集合没有发生改变，所以迭代开始时，v_1的 PR 值不变，而v_2新增指向它的顶点，所以它的 PR 值在原来的基础上增加v_1顶点的贡献值，即

$$PR(v_2) = PR(v_2) + d * \frac{PR(v_1)}{|N_+(v_1)|}$$

其中d为调整因子，$N_+(v_1)$为v_1指向的邻接点的集合，$|N_+(v_1)|$为v_1指向的邻接点的集合的元素数目。

虽然v_1的 PR 值不变，但是v_1对其指向的邻接点的贡献发生了变化，所以需要以v_1, v_2为中心，将这种变化传播到它们指向的邻接点。

②UPDATE 事件

如果新流入系统的边，只是修改原图中的边的权重，则不会影响图中各个顶点的 PR 值，即当事件类型为 UPDATE 时，图的状态保持不变。

③DELETE 事件

如果新流入系统的边，是希望在原图中将这条边删除，即从右往左看图 4-2(d)，此时对于v_1顶点，指向v_1的顶点集合没有发生改变，所以迭代开始时，v_1的 PR 值不变；对于v_2顶点，由于删除了v_1指向v_2的这条边，所以v_2的 PR 值将在原来的基础上减少v_1顶点对其的贡献，即

$$PR(v_2) = PR(v_2) - d * \frac{PR(v_1)}{|N_+(v_1)|}$$

类似在① ADD 事件的分析过程，虽然v_1的 PR 值没有发生改变，但是由于删掉了v_1指向v_2的这条边，导致v_1对其指向的其它邻接点的贡献发生了变化，因此需要以v_1, v_2为中心，将这种变化传播到它们指向的邻接点。

在分析完流式场景下的算法细节后，下面本文将详细定义算法的三个组件：

（1）State：图的 State 由每个顶点对应的邻接点的信息组成，$State = \{s_1, s_2, \dots, s_n\}, n = |V|$，其中$s_k = (v_k, pr_k)$，表示顶点$v_k$的 PR 值为$pr_k$；

（2）Event：图的 Event 为图到达一条边相关的事件，Event 构成的序列形成事件流，即$Event\ Stream = z_1, z_2, \dots, z_m$，其中$z_k = (e_k, TYPE)$，$TYPE \in \{ADD, UPDATE, DELETE\}$，这三种状态对应的事件分别为增加一条边，更新一条边和删除一条边；

（3）Transform：图的状态在事件流的驱动下的转换函数，如算法 5 所示：

算法 5 Dynamic PageRank

```
1   for each z ∈ Event Stream
2       do (v1,v2) ← GET-VALUE(z)
3           type ← GET-TYPE(z)
4           s1← GET-STATE(v1)
5           s2←GET-STATE(v2)
6           pr1← GET-STATE(s1)
7           pr2←GET-STATE(s2)
8           if s1= ∅ and s2= ∅
9               then SET-STATE(v1, (1 − d)/(|V| + 2))
10                   SET-STATE(v2, (1 − d)/(|V| + 2) + d)
11          elseif s1≠ ∅ and s2= ∅
12              then SET-STATE(v2, 1)
13                   SPREAD-TO-OUT-NEIGHBOR(v1)
14          elseif s1=∅ and s2≠ ∅
15              then SET-STATE(v1, (1 − d)/(|V| + 1))
16                   SET-STATE(v2, pr2 + d)
17                   SPREAD-TO-OUT-NEIGHBOR(v2)
18          else
19              if type = ADD then SET-VALUE(v2, pr2 + d ∗ pr1/|N+(v1)|)
20              elseif type = DELETE then SET-VALUE(v2, pr2 − d ∗ pr1/|N+(v1)|)
21              else return
22              SPREAD-TO-OUT-NEIGHBOR(v1)
23              SPREAD-TO-OUT-NEIGHBOR(v2)
```

算法 5 中，第 2-7 行是获取源顶点v_1和目标顶点v_2的 PR 值；8-10 行对应（a）情况，当两个顶点都是新顶点时，设置它们的初始值分别为$(1-d)/(|V|+2)$和$(1-d)/(|V|+2)+d$；11-13 行对应（b）情况，当目标顶点是新顶点，源顶点是已经存在于原图中的顶点时，首先设置新顶点的值为 1，然后以源顶点为中心，向它所指向的邻接点传播影响；14-17 行对应（c）情况，当源顶点为新顶点，目标顶点为已经存在于原图中的顶点时，首先更新源顶点和目标顶点的值，然后以v_2为中心向它所指向的邻接点继续传播影响；19-21 行是根据事件的类型对目标顶点的 PR 值进行更新；22-23 行是以v_1和v_2为中心，向它们的邻接点传播这种影响。

SPREAD-TO-OUT-NEIGHBOR(v)函数是传播函数，它将触发v向其指向的邻接点发送信息，这个信息就是v对其指向的邻接点的贡献值，接收到v所发信息的邻接点，将重新计算其 PR 值，如果其 PR 值发生变化，则它又会向其指向的邻

接点发送消息，依次传播下去，直到所有顶点的 PR 值不再发生变化或者变化范围在某个阈值内，则此时算法运行结束。

4.5 本章小结

本章重点分析了 DD、TC、SSSP 和 PR 算法在流式场景下的设计细节，并且结合第三章建立的基于状态更新传播的流式图计算模型，阐述了如何对这四个算法进行设计。这四个算法只是图算法中较为典型的算法，本文旨在说明如何利用第三章提出的基于状态更新传播的流式图计算模型来进行算法设计，读者可根据这四个算法的设计细节，将图算法中其它算法设计成流式图算法。

第五章 GraphFlow 系统的设计与实现

在本章节，设计并实现了基于状态更新传播的流式图计算模型—GraphFlow 系统。本节将从系统架构、模型和算法实现三个层面阐述系统内部组件的实现细节以及系统的运行过程。

5.1 系统架构

GraphFlow 系统是面向流式图数据的实时图计算系统，它根据原始图的状态和增量图数据进行增量式的更新，当连续不断的图数据流入系统后，它将图数据流转换成事件流，再由事件驱动图的状态不断发生改变。相比较基于 BSP 模型的批处理系统，GraphFlow 通过增量计算的形式，充分利用原始计算结果和增量图数据来进行计算，系统的实时性更强；相比较基于估计模型的流处理系统，GraphFlow 通过变化传播的方式来迭代计算，提高了计算结果的准确性。GraphFlow 系统的架构图如图 5-1 所示。

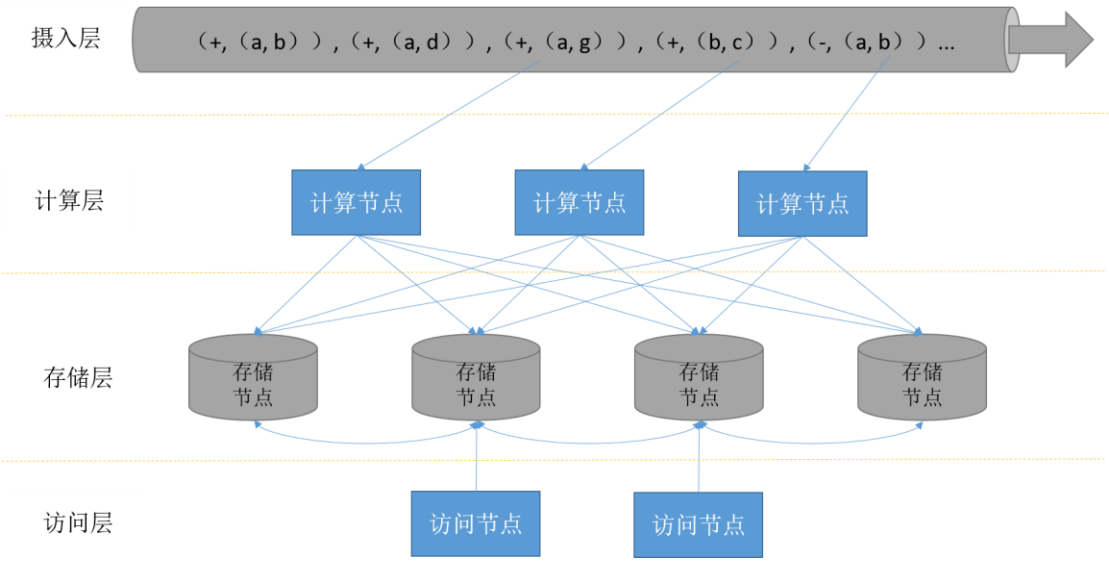


图 5-1 GraphFlow 系统架构图

从整体上看，GraphFlow 系统可以分为以下 4 个部分：

- 摄入层：GraphFlow 系统的输入，图数据以流的形式流入到系统的各个计算节点中。这些图数据可以以文件的形式存储，也可以存储在如 Kafka，HBase 等其它分布式系统中。GraphFlow 系统提供了接口，能够很好的将这些数据源对接到系统。
- 计算层：GraphFlow 系统的核心层。摄入层提供的数据将分配到各个计算节

点， 计算节点可以访问存储层中的图状态，并且根据图的状态和当前接收的事件来触发图状态的更新，并且将这种更新同步到存储节点中，以便其它计算节点能够立刻使用。

- 存储层：负责整个系统的状态的存储。该层采用分布式存储架构，系统内的状态分散到各个存储节点上进行存储和备份，同时还提供了持久化接口，也可以将这些状态异步备份到永久性介质上，进一步提高系统的可靠性。由前文可知，状态是从用户的视角进行定义的，直接反应了用户关心的数据，所以在同一时刻，系统可能存在多种不同类型的状态，这些状态对系统内部可以由计算节点直接访问，对系统外部也可以由用户根据访问节点，实时访问中间计算结果。在本层本文利用开源产品内存数据网格 Hazelcast[50]来作为存储层，存储图的状态信息，Hazelcast 提供了细粒度的分布式锁结构，本文使用该接口来锁住顶点的状态。
- 访问层：向最终用户提供接口，允许用户在任意时刻访问图的状态。在本层本文使用 RESTful 规范来设计数据的访问规则，利用 Jetty 作为内嵌的服务器，向用户提供数据访问能力。

这四层涉及图数据的输入、计算、存储和访问，是流式图数据完整的处理过程。而中间的计算层是图计算的核心，下面将详细讨论这层的设计。

计算层的设计如图 5-2 所示，该层根据摄入层接收的图数据和存储层存储的当前图状态，进行增量式的计算，根据计算结果对图的状态及时更新，而且这种更新能够立刻被其它计算节点感知。该层又可以细分为：应用层、服务层、API 层和核心层。

<table>
<tr><td>应用层</td><td colspan="2">Application</td></tr>
<tr><td>服务层</td><td colspan="2">Library</td></tr>
<tr><td>API层</td><td colspan="2">Graph Streaming</td></tr>
<tr><td rowspan="2">核心层</td><td>Graph</td><td>Streaming</td></tr>
<tr><td colspan="2">Computing Model</td></tr>
</table>

图 5-2 计算层设计图

每层的功能为：

- 应用层：面向用户的上层运用，这些运用涵盖了典型的业务场景，例如链接分析、欺诈检测、社区发现等，是针对某个具体问题的具体应用。该层组合了服务层提供的各种库函数，为特定的业务场景定制解决方案。一般来说由用户来实现。
- 服务层：提供给用户使用的丰富的库函数和图算法。如前文所述的 DD、TC、SSSP 和 PR 算法，都在该层提供。

- API 层：该层屏蔽了底层的实现细节，向用户提供了一个统一的流式图数据的处理接口，用户可以组合这些接口完成特定的流式图计算。
- 核心层：核心层抽象出图和流的概念，并且通过一个统一的计算模型将两者结合起来。采用基于状态更新传播的流式图计算模型作为计算模型，采用面向用户视角的状态作为编程模型，提供了实时图计算的能力。

5.2 模型实现

在第三章中，本文详细介绍了基于状态更新传播的流式图计算模型。由前述所知，该模型的三个核心组件：State，Event，Transform 是实现模型的关键。下面将重点介绍这三个组件的实现。

5.2.1 状态（State）

状态反应了图当前的特征信息，这些特征信息是以用户的视角来进行定义的，只存储用户关心的图的某些特征信息。一般而言，这些特征信息是以顶点为单位进行组织的，即图中每个顶点的状态的集合组成了整个图的状态，本文将这种状态称之为离散状态（Individual State）；此外，图的特征信息还可能以单独一个值来体现，如希望统计图的顶点数目，此时用一个计数器就可以表示图的状态，本文将这种状态称之为聚合状态（Integral State）。状态类图如图 5-3 所示。

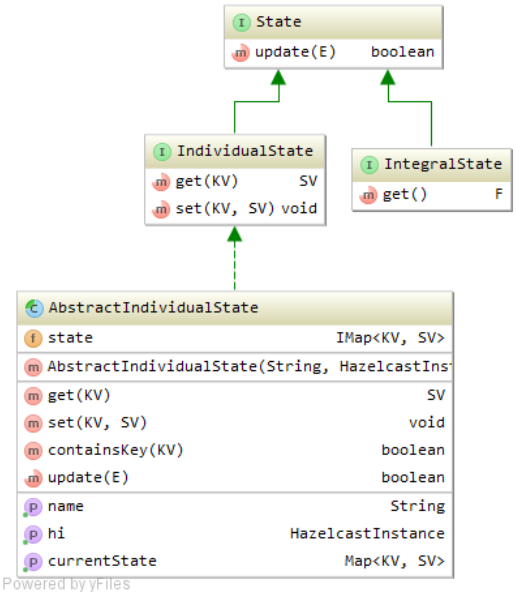


图 5-3 State 接口和抽象类设计图

State 组件在包 com.duansky.hazelcast.graphflow.state 中，它们各自的主要功

能如下：

- State：所有状态的父接口。它只有一个更新函数 update(E)，即 接收的事件对图的状态进行更新。
- IndividualState：离散状态接口。它表明状态是由一个个的离散的顶点状态组成，用户可以通过 set(KV,SV)设置某个顶点的状态，也可以通过 get(KV)来获取某个顶点的状态。
- IntegralState：聚合状态接口。它表明状态通过一个单独的值来表明。用户可以通过 get()来获取状态。
- AbstractIndividualState：离散状态的抽象类。它实现了离散状态 get(KV),set(KV,SV),containsKey(KV)等较为通用的函数，为离散状态的具体实现类提供了模板。

在 GraphFlow 系统中，本文实现了很多状态子类，如图 5-4 所示：

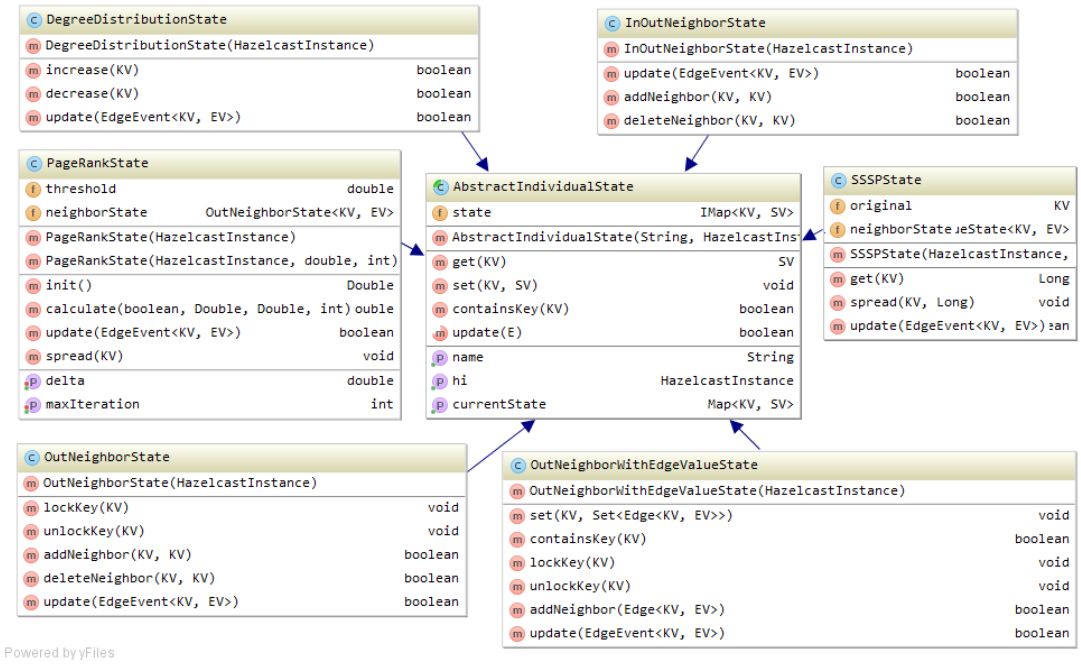


图 5-4 State 详细类图

按照用途的不同，可以将这些实现子类分为基础状态类和算法状态类，基础状态类是用来记录图的基本信息，这些基本信息可能反复被其它的算法所使用，而算法状态类就对应于某个具体的算法，它保存了该算法在计算过程中的中间状态。

基本状态类有如下几种：

- OutNeighborState：该类用来记录某个顶点所指向的顶点的集合，采用$<vertex, N_+(vertex)>$键值对的形式来存储数据，其中$vertex$是顶点的标号，$N_+(vertex)$是顶点所指向的顶点集合。

- InOutNeighborState：该类用来记录某个顶点所指向的顶点集合以及指向它的顶点的集合，采用$< vertex, (N_{-}(vertex), N_{+}(vertex)) >$键值对的形式存储数据，其中$vertex$是顶点标号，$N_{-}(vertex)$是所有指向该顶点的顶点集合，$N_{+}(vertex)$是该顶点指向其它顶点的顶点集合。相比较 OutNeighborState 类，它不仅记录了顶点所指向的邻接点信息，还记录所有指向该顶点的邻接点信息。
- OutNeighborWithEdgeValueState：该类用来记录某个顶点所指向的顶点以及这两个顶点所组成的边的值的集合，采用$< vertex, N_{+}((vv, ev)) >$键值对的形式来存储，其中$vertex$是顶点的标号，$vv$是$vertex$所指向的顶点，$ev$是$(vertex, vv)$所组成的边的值，$N_{+}$是所有这样$(vv, ev)$对的集合。相比较 OutNeighborState 类，它不仅记录了顶点所指向的邻接点标号，还记录了它们所组成的边的值。

算法状态类是根据具体的算法而设计的，这样设计的目的是将不同算法的计算结果单独存储，这样用户可以同时在系统中执行不同的算法，而且分别获取各自算法的执行结果，互不干扰；而且用户只需要根据系统提供的接口就可以很快设计并实现新的算法。关于算法状态类的实现因为是跟算法紧密相关的，本文在算法实现章节再做具体介绍，这里不再阐述。

5.2.2 事件（Event）

基于状态更新传播的流式图计算模型将连续流动的图数据流抽象成一个个的事件（Event），再由这些事件来触发图状态的转换。因此本质上讲，事件反应了图数据的变化。

事件由事件类型（Event Type）和事件值（Event Value）组成，事件类型反映了图数据的操作类型，即新流入系统的数据，相比较原来的图数据是增加（ADD）、更新（UPDATE）还是删除（DELETE）；事件值反映了这条图数据是边相关的图数据（包括边的源顶点，目标顶点和边的权重）还是顶点相关的图数据（包括顶点的标号和顶点的值）。连续不断的事件流入系统，就形成了事件流（Event Stream）。事件相关的接口和类设计见图 5-5。

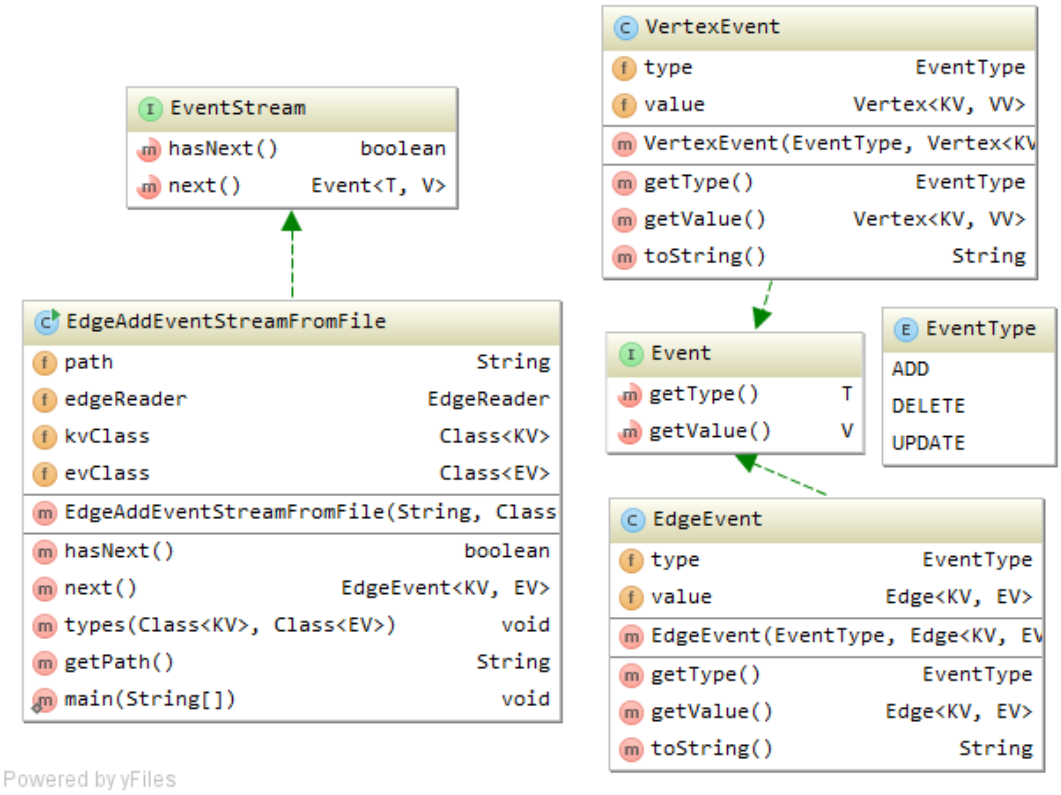


图 5-5 Event 接口和类设计图

Event 组件在包 com.duansky.hazelcast.graphflow.event 中，它们各自的主要功能如下：

- Event：所有事件类的父接口。它定义了一个事件应该具有的两个基本方法：获取事件的类型 getType()和获取事件的值 getValue()，用户可以实现该接口来定制特定类型的事件。
- EventType：定义了基本事件的类型，在本系统中，基本事件类型有 ADD，UPDATE 和 DELETE。
- EdgeEvent：定义了边相关的事件，即事件中的值是一条边相关的信息，包括边的源顶点，目标顶点和边的权重。
- VertexEvent：定义了顶点相关的事件，即事件中的值是一个顶点相关的信息，包括顶点的标号和顶点的值。
- EventStream：将连续不断的进入系统中的事件抽象成一个事件流。它提供两个基本方法：（1）hasNext()用来判断是否到达流的末尾；（2）next()从事件流中取出一条事件。EventStream 本质上是对数据源进行抽象，使得用户实现该接口后就可以和特定的外在系统进行对接。

5.2.3 转换（Transform）

转换是由事件触发的图的更新过程，是流式图计算模型中的计算逻辑，详细定义了图如何根据到达的事件，从一个状态转变成另外一个状态。因为转换是跟具体的状态相关的，转换离不开状态而单独存在，它是状态固有的一个方法，所以在实现的时候，本文将转换作为状态的一个函数，即状态的 update()函数。在后续的版本中，我们将会考虑将转换剥离开来，但这只是实现上的区别，不会影响整个系统的执行过程。

5.3 算法实现

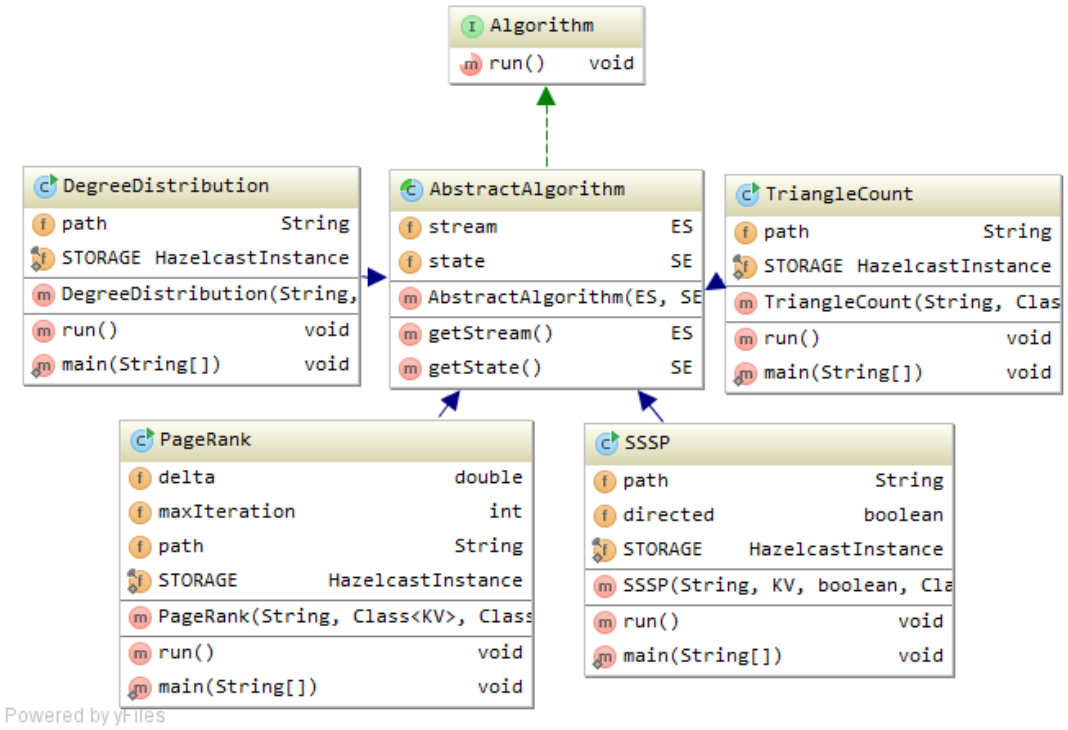


图 5-6 算法类图

图 5-6 是 GraphFlow 系统提供的算法相关的接口和类。算法在包 com.duansky.hazelcast.graphflow.lib 中，它们各自的主要功能如下：

- Algorithm：所有算法必须实现的接口。Algorithm 接口只有一个 run()方法，该方法是算法的核心处理逻辑。
- AbstractAlgorithm：算法的抽象类，该抽象类有两个重要属性：事件流和状态。事件流是该算法需要处理的事件流，其定义在 5.2.2 节中已经详细阐述；状态是算法在增量计算过程中需要存储的中间状态，一般情况下，每个算法都

至少有一个状态与之对应。

在定义完接口和抽象类后，下面定义了四个具体的算法实现类，这四个类分别对应四个流式图算法。它们都继承自 AbstractAlgorithm，并且实现了核心的 run() 方法。由于篇幅限制，本文选取 TC 和 SSSP 这两个算法来讲解算法的实现过程，以帮助用户能够利用该系统来实现特定的算法。

5.3.1 三角形计数算法

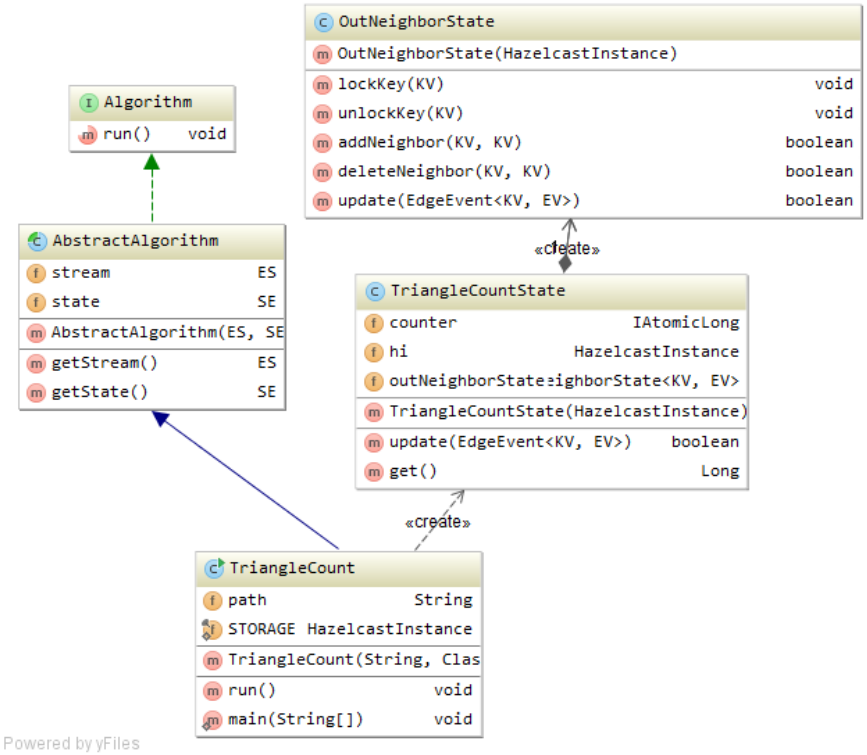


图 5-7 Triangle Count 算法相关类图

图 5-7 详细展示了 TC 算法实现相关的类。TC 算法继承自 AbstractAlgorithm 抽象类，并且实现了 run() 方法；同时 TC 算法有一个核心的数据结构 TriangleCountState，它用来存储 TC 算法的中间计算状态。

算法 6 RUN()

```
1    public void run(){
2        while(stream.hasNext()) {
3            EdgeEvent<KV, EV> edgeEvent = stream.next();
4            state.update(edgeEvent);
5        }
6    }
```

run()函数的处理逻辑也比较简单，如算法 6 所示，它从事件流中取出一个事

件，然后将这个事件提交给状态，由状态的 update()函数来根据事件进行增量式的更新。

状态是如何根据事件进行更新的呢？其设计思路本文在 4.2 节 TC 动态算法设计中已经描述得非常详细，即根据新增边的源顶点和目标顶点的公共邻接点的数目和事件的类型来更新图的 TC 值。注意到 TriangleCountState 类还利用了基础类 OutNeighborState，状态是可以进行组合和重复利用的，这样使得程序的复用度高，而且共用相同的状态也可以减少数据存储空间。

5.3.2 单源点最短路径算法

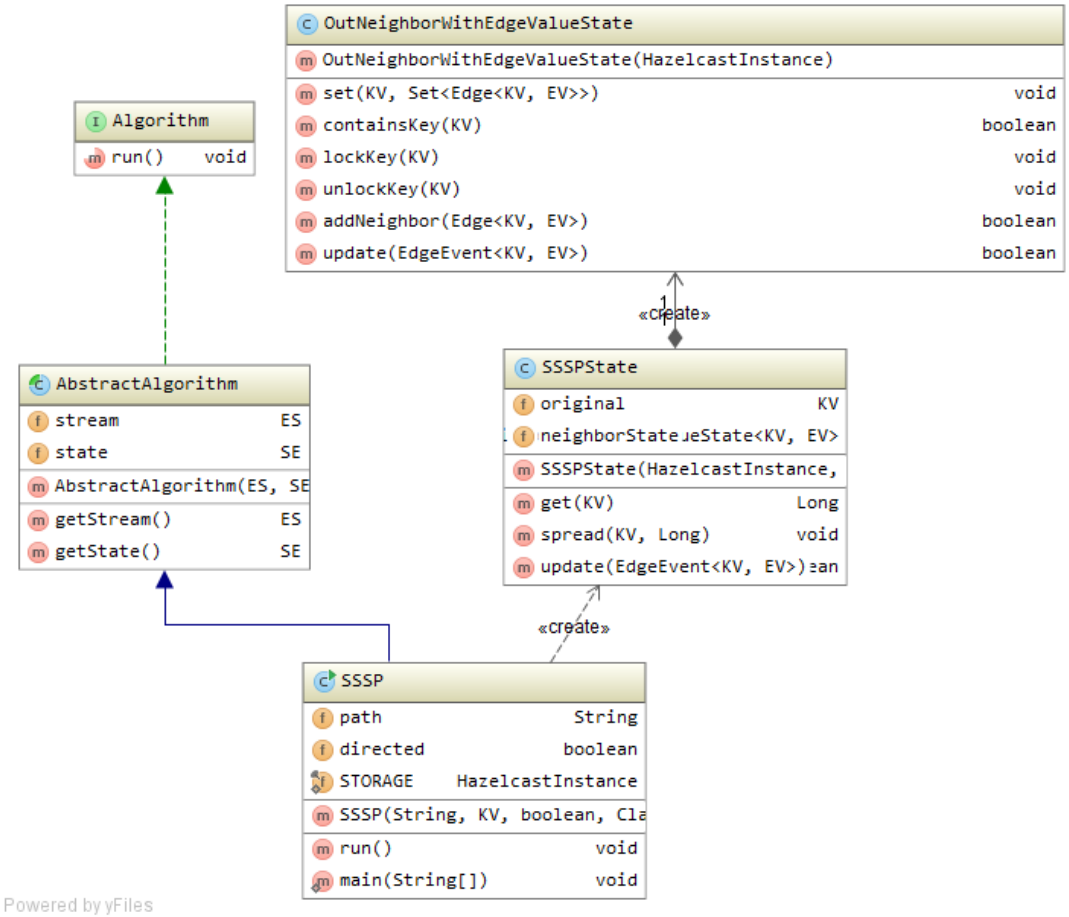


图 5-8 SSSP 算法相关类图

图 5-8 详细展示了 SSSP 算法实现相关的类。类似于 TC 算法，SSSP 算法也是继承自 AbstractAlgorithm 抽象类并实现了 run()方法；SSSP 算法也有一个核心的数据结构 SSSPState 用来存储 SSSP 算法的中间计算结果。下面重点讲解一下在 SSSP 算法中，顶点的影响是如何传播给其它顶点的。

算法 7 SPREAD-TO-OUT-NEIGHBOR(VERTEX_ID,VALUE)

```
1       for(Edge<KV,EV> edge : neighbors){
2           target = edge.getTarget();
3           if(state.containsKey(target)){//if this vertex has already calculated.
4               tarOldValue = get(target).longValue();
5               tarNewValue = value + edge.getEdgeValue().longValue();
6               if( tarNewValue < tarOldValue){ // if the new value is smaller.
7                   spreadToOutNeighbor(target,tarNewValue);
8               }
9           }else{//else the vertex is reachable now.
10              tarNewValue = value + edge.getEdgeValue().longValue();
11              spreadToOutNeighbor (target,tarNewValue);
12          }
13      }
```

算法 7 的目的是遍历顶点所有邻接点，并且检测是否需要去更新它们的 SP 值。第 2 行用来获取该顶点的一个邻接点；第 3 行判断这个邻接点是否存在对应的状态（即 SP 值）；4-8 行用来处理该邻接点已经有对应的 SP 值的情况，此时需要考虑该顶点到达这个邻接点是否是更近的，如果是，则更新这个邻接点的状态，同时该邻接点需要继续传播影响，否则不做任何操作；10-11 行用来处理盖邻接点还没有对应的 SP 值的情况，那么该邻接点更新后的值即为该顶点的值加上边上的权重，更新这个邻接点的状态后，同样需要继续传播这个邻接点对其邻接点的影响，因此继续调用 spreadToOutNeighbor()函数。

5.4 本章小结

本章首先介绍了面向流式图数据的实时图计算系统 GraphFlow 的系统架构，从摄入层、计算层、存储层和访问层四个层面描述了系统的整体布局，并且说明每层的功能和实现方式；随后重点介绍了系统的模型实现，按照基于状态更新传播的流式图计算模型的三个组件：状态、事件和转换逐一进行解释说明；并且以 TC 算法和 SSSP 算法为例，讲解了如何使用这三个组件来实现特定的算法。

第六章　实验与分析

在本章节，本文将搭建实验环境来测试 GraphFlow 系统性能。首先介绍了实验所采用的数据集规模和硬件环境，随后从准确性、实时性和更新冲突概率三个层面对系统进行了测试，并且给出实验结果和实验分析。

6.1 实验环境

本文采用了斯坦福大学提供的 Live Journal[51]的社交数据作为实验数据，Live Journal 是一个拥有上千万用户的综合性的交友网站，用户可以记录日志、游记和撰写博客。为了能够实测在不同数据集上的系统性能表现，本文采用平均抽样的方式，将原始的数据集抽样形成 10 组不同规模的数据集，尽量保证这 10 组数据集的规模按线性排列。测试数据集见表 6-1 所示，其中 D10 为原始的全集数据，总共约有 480 万个顶点，6900 万条边；D1-D9 为通过平均抽样方式从 D10 数据集抽样组成。

表格 6-1 测试数据规模表

编号	顶点数目	边数目
D1	1,459,523	6,899,377
D2	2,091,376	13,798,754
D3	2,580,120	20,698,131
D4	2,909,471	27,597,508
D5	3,175,063	34,496,885
D6	3,439,002	41,396,262
D7	3,726,902	48,295,639
D8	3,988,881	55,195,016
D9	4,331,528	62,094,393
D10	4,847,571	68,993,773

数据来源: http://snap.stanford.edu/data/soc-LiveJournal1.html

为了快速了解数据集的特征，本文首先在全集数据（D10 数据集）上统计了顶点的度的分布图，如图 6-1 所示，由此可见大多数的顶点的度都集中在 1-10 之间，只有极少数的顶点的度能够超过 30。

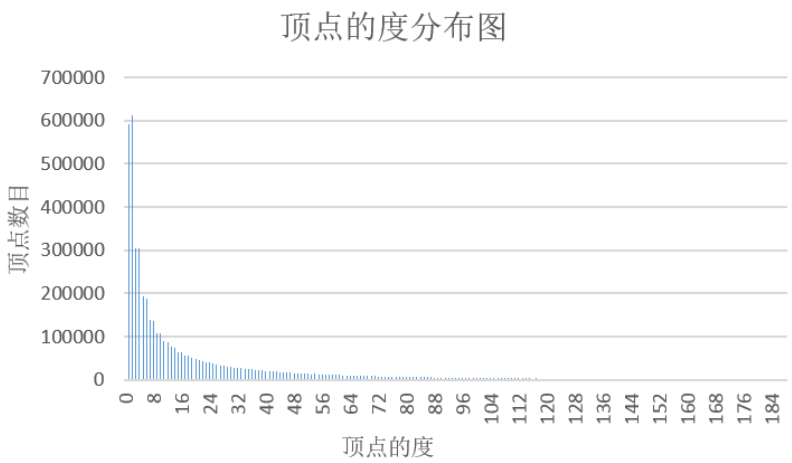


图 6-1 顶点的度分布图

本实验主要通过测试 DD、TC、SSSP 和 PR 这四个算法进行验证。实验在由 10 台计算机构成的集群上运行。每台计算机从本地读取文件并进行计算，然后更新分布式缓存中的图的状态。单独每台计算机的配置如表 6-2 所示：

表格 6-2 计算节点配置表

参数	配置
处理器	8 核 Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz
内存	16G RAM
硬盘	2 * 1TB SATA
操作系统	Ubuntu 11.04

6.2 实验结果

本文提出的基于状态更新传播的流式图计算模型，相比较传统的 BSP 模型，能够实时计算并反馈中间结果；相比较流式图计算的估算模型，本模型采用准确计算的方式能够提高计算结果的准确率；相比较 IncGraph 的串行更新模型，本模型采用基于细粒度分布式锁的方式实现了并行更新，而且锁冲突的概率很小，因此本实验从实时性、准确性和更新冲突概率三个方面来进行了测试和验证。

6.2.1 实时性

相比较传统的批处理图计算模式，GraphFlow 系统最为显著的特点就是支持实时图计算。为了详尽展示不同算法的实时计算能力，本文分别测试了 DD、TC、SSSP 和 PR 算法在全集数据 D10 上的实时计算能力。图 6-3 至图 6-6 分别展示了这四个算法的实时性，左图展示的是算法的累计分布图（Cumulative

Distribution Function，CDF），右图展示的是每个响应时间（处理一条增量数据所需的时间）所占的百分比图。

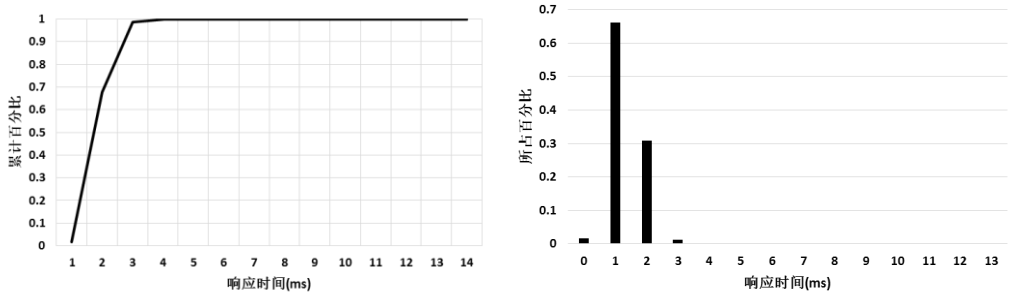


图 6-2 DD 算法实时性图

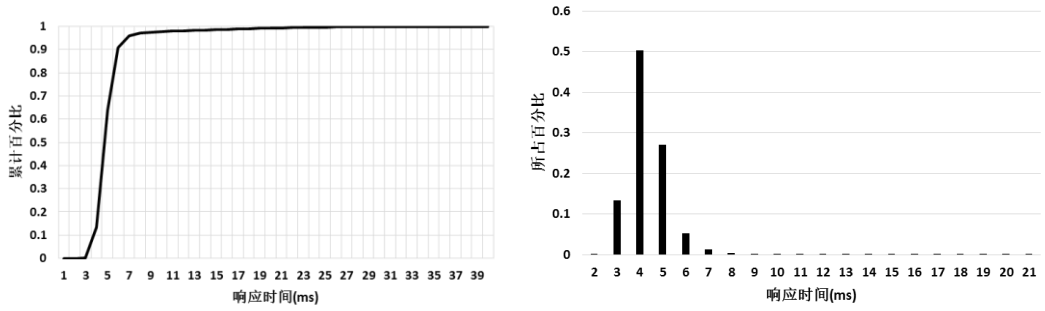


图 6-3 TC 算法实时性图

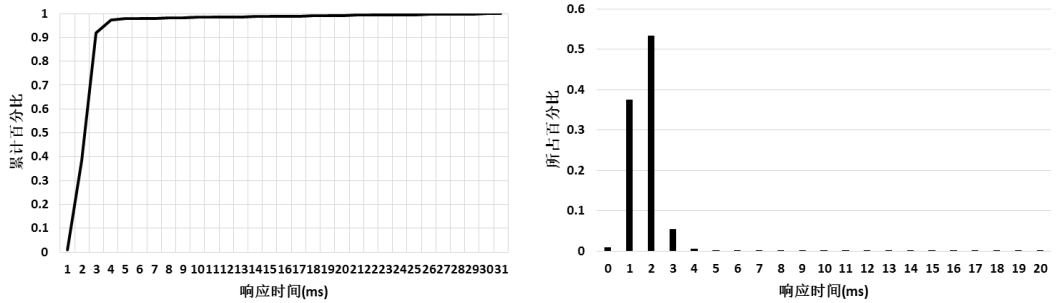


图 6-4 SSSP 算法实时性图

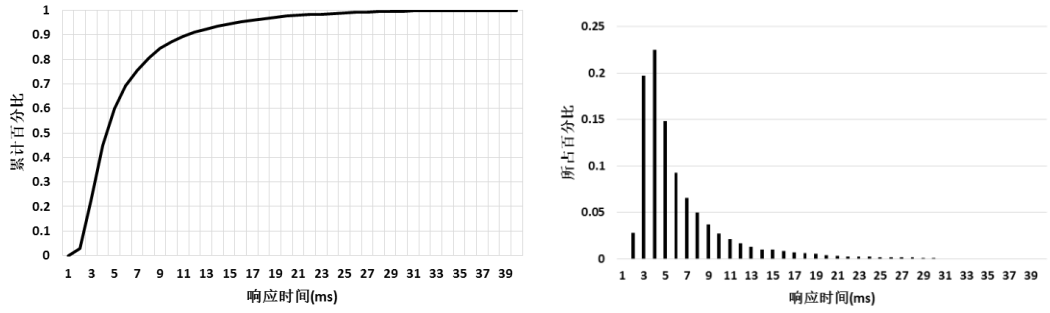


图 6-5 PR 算法实时性图

从算法的 CDF 图来看，这四个算法的 90%的请求都能够在 12ms 内立即得到响应，符合实时性的要求；从算法的实时性分布图来看，不同算法的响应时间

略有不同，但整体的响应时间分布符合长尾效应：（1）DD 算法的平均响应时间最短，超过一半的更新请求都能够在 1ms 的时间内得到响应；（2）TC 算法和 SSSP 算法的平均响应时间比 DD 算法要长，而且大部分请求的响应时间都集中在 2ms 或 4ms；（3）PR 算法的平均响应时间最长，但 90%的请求都能够在 12ms 内得到响应，而请求的响应时间大部分都集中在 4ms-8ms 之间。

不同算法的更新代价之所以不同，是因为不同算法所影响的顶点的数目不同。如对于 DD 算法，每次到达的边数据只会影响这条边的源顶点和目标顶点，而且这种影响不会传播给其它的顶点，所以响应时间最短；对于 TC 算法，它不仅影响这条边的两个顶点，还会影响这两个顶点的所有公共邻接点；对于 SSSP 算法，它会以源顶点为中心，将这种影响按照往外扩散的路径将影响传播开来；而对于 PR 算法，在极端情况下它可能影响到整个连通子图，所以响应时间最长。因此，增量数据对不同算法所带来的更新的影响范围是不同的，这种不同直观反映在了更新请求的响应时间。此外，本文还发现尽管不同算法的响应时间不同，但其分布均符合长尾效应。这是因为自然状态下的图数据分布是符合 power-law 规则的，而符合 power-law 规则的图的顶点分布即为长尾分布。这也使得有极少数的顶点拥有大量的邻接点，它们的更新代价很高，而大部分顶点的邻接点都很少，所以更新得很快，因此大多数顶点的响应时间都很短，只有极少数顶点的响应时间很长。

6.2.2 准确性

GraphFlow 系统采用了增量计算的方式来实时处理流式图数据，相比较传统的基于采样和概要设计的估算模式，计算结果更为准确。为了能够测试出系统并发度对系统计算结果准确性的影响，本文分别测试了 DD、TC、SSSP 在并发度为 1-10 的情况下，计算结果的准确性，因为 PR 算法现有的系统实现方式不同，得到的计算结果也不相同，无法对测试结果进行很好的评判。为此，本文重点关注前三种算法的准确性，对于 PR 算法，重点关注它的实时性。

DD 算法用来统计每个顶点的度，SSSP 算法用来计算各个顶点到源点的最短距离，对于这两个算法，本文的正确率计算公式为：

$$P = \frac{\sum g(v)}{|V|} * 100\% \qquad \text{（公式 4）}$$

其中$g(v) = \begin{cases} 1, & v\text{值计算正确} \\ 0, & v\text{值计算错误} \end{cases}$。即将比对每个顶点实际计算的值跟真实值是否相同，计算正确的顶点所占的比例即为该算法的准确率。

而对于TC算法，它反馈回来的只有一个单独的值，即整个图的三角形数目。对于该算法，正确率计算公式为：

$$P = \left(1 - \frac{|TC_{calculate} - TC_{real}|}{TC_{real}}\right) * 100\% \quad （公式 5）$$

其中$TC_{calculate}$为本系统中的 TC 算法计算出来的值，TC_{real}为图真实的 TC 值。

定义完这三个算法的准准确率计算公式后，本文在 D1-D10 数据集上分别进行测试，同时为了考虑不同并发度对计算结果准确性的影响，本文分别测试了这三个算法在计算节点总数为 1、2、4、6、8、10 下的准确率，实验结果如图 6-2 所示。

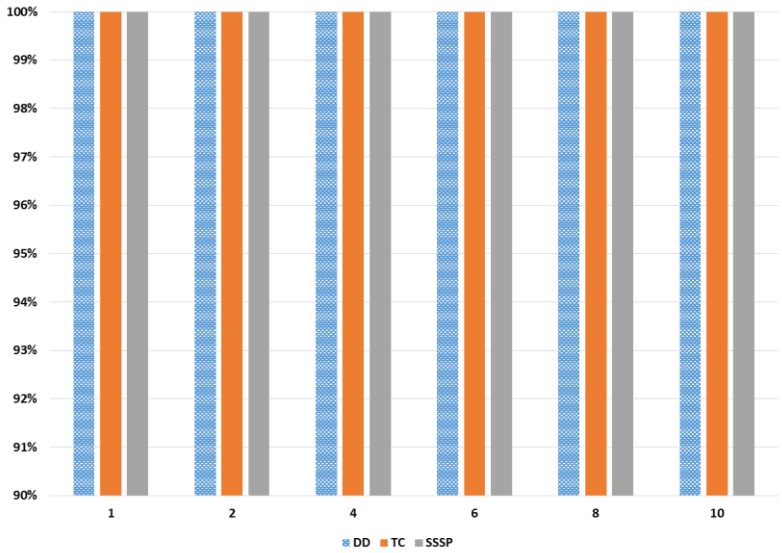


图 6-6 算法准确率测试结果图

这三个算法的准确率在不同并发度下均为 100%，是因为在实现时本文采用了基于细粒度分布式锁的更新策略，保证了算法最终计算结果的准确性。

6.2.3 更新冲突概率

在 3.3 节关联状态的存储和更新时，本文提出采用基于细粒度分布式锁的并行更新策略，这种策略使得不同计算节点可能会竞争同一个顶点的状态，出现更新冲突现象。为了评估这种更新冲突的概率，本文设计了本实验。

实验架构图见图 6-7 所示。本实验中本文首先将全集数据 D10 按照边的源顶点和目标顶点的大小顺序排序，然后将 D10 数据集采用 Round-robin 的方式分配给 10 个计算节点分别进行计算。这样做的目的是通过对边进行排序和按序发送给各个计算节点来最大限度地提高计算节点之间发生更新冲突的概率。

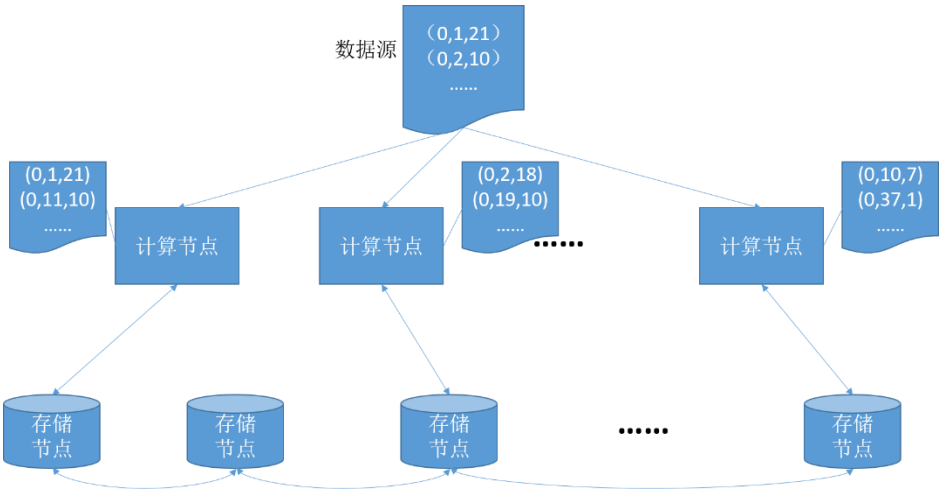


图 6-7 更新冲突检测实验架构图

同样为了检测不同算法的冲突概率，本文分别测试的 DD、TC、SSSP 和 PR 算法，它们的实验结果如图 6-8 所示。

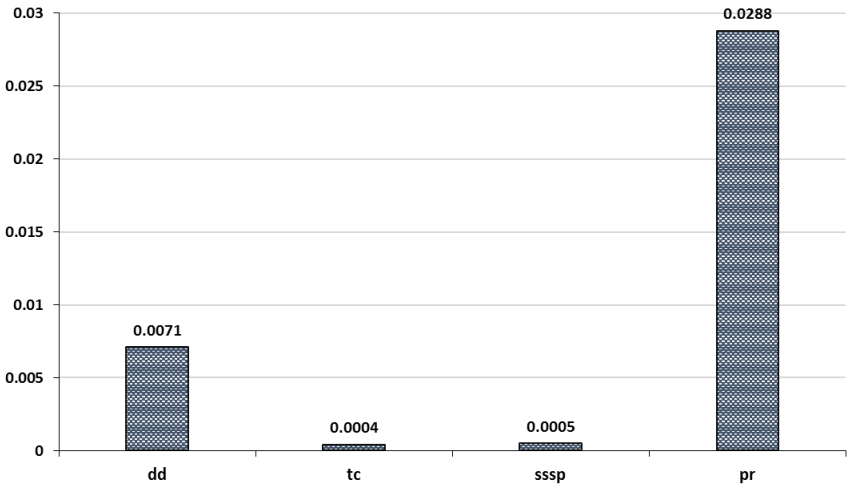


图 6-8 算法更新冲突概率图

由图 6-8 可以看出，这四个算法的更新冲突概率都在 3%以下，每个算法的更新冲突概率略有不同。整体上来看，PR 算法更新冲突概率最大，这是因为 PR 算法每次更新时，会以新增的边的两个顶点为中心向外传播，可能会影响整个连通子图内的所有顶点，所以更新冲突概率最大；而 SSSP 算法是按照以新增边的源顶点为中心，沿着某条路径往外传播的方式，有可能传播到某个顶点就结束（当新增的这条边不能使得其它邻接点的 SP 值减小时传播结束），因此传播的影响范围没有 PR 算法大，所以更新冲突的概率相比 PR 算法会小很多；TC 算法只会影响新增边的两个顶点的所有公共邻接点，影响范围更小，更新冲突的概率就更小；至于 DD 算法，按照影响范围来看应该是最小的，但是由于本实验的人为设计，即将顶点编号相近的边尽量安排在不同的计算节点上，这将导致多个计算节点同时争取更新同一个顶点的状态，所以更新冲突的概率也比较高，如果在自然

分发的情况下，因为 DD 算法的更新只会影响到新增这条边的两个顶点，影响范围最小，所以更新冲突概率是最小的。

6.3 本章小结

本章通过实验的方式，验证了 GraphFlow 系统的准确性、实时性和更新冲突概率。实验结果表明，采用基于状态更新传播的流式图计算模型，系统的 90%更新请求都能够在 12ms 内得到响应；在不同并发度下系统仍能保持较高的准确率；采用细粒度分布式锁来控制多个节点的并发更新，发生更新冲突的概率小于 3%。

第七章 结束语

在本章节，先对本文的工作做了一个总结，随后对进一步的改进方向做出几点展望。

7.1 工作总结

本文首先阐述了现阶段图数据和图算法的特点，并总结了现有的图计算相关的关键技术和常见的图计算模型及系统，在归纳总结图算法特征之后，针对这些特征提出了基于状态更新传播的流式图计算模型，设计并实现了基于该模型的GraphFlow 系统，并且给出了系统测试结果及实验分析。具体来讲，本文的主要工作内容有以下几个方面：

- 分析了现阶段图数据海量和动态特性，图算法局部性差和迭代计算的特点，并且从图的划分、编程模型和计算泛型三个方面总结了解决海量图数据所需的关键技术，同时按照处理图数据的不同，从批处理和流处理两个角度出发，列举并分析了现有的常见的图计算模型和系统。
- 以 DD、TC、SSSP 和 PR 算法为代表，详细分析了这四个算法在流式图数据场景下所呈现的特征，从影响范围、计算方法、计算顺序、计算性质和计算次数这五个层面来详细展示每类算法的特点，并且总结了能够适用于流式场景下的图算法应该具备的三个典型特征：（1）计算方法满足增量计算特性；（2）计算顺序满足序列一致性；（3）计算性质满足代数运算的交换律和结合律。
- 结合前文对能够适用于流式场景下的图算法所具备的典型特征分析，设计了基于状态更新传播的流式图计算模型，该模型将动态变化的图数据抽象成连续不断的事件流，将图的中间计算结果抽象成图的状态，根据前一时刻的历史状态和到达的事件，以增量计算的方式来触发图状态的更新。该模型能够有效解决流式图计算的问题，而且通过增量计算的方式，既减少了每次状态更新的代价，又使得计算的结果相比估算模式更加准确。
- 根据前文提出的基于状态更新传播的流式图计算模型，设计并实现了GraphFlow 系统。本文不仅给出了 GraphFlow 系统实现计算模型的细节，而且还给出了实例帮助用户快速在该计算模型上实现流式图算法。最后通过真实的数据集对系统进行了实时性和准确性测试，测试结果表明系

统的准确率达到 99%以上；90%以上的请求都能够在 12ms 内返回，符合实时性要求；采用细粒度分布式锁来控制多个节点的并发更新，发生更新冲突的概率小于 3%。

7.2 下一步工作

由于时间有限，本文所设计的流式图计算系统仍有很多不足之处。下一步工作可以就以下几个方向展开：

- 模型方向：本文是以 DD、TC、SSSP 和 PR 算法为例，分析了它们在流式场景下所具备的增量计算，序列一致和满足交换律和结合律三个特征，进而抽象出了基于状态更新传播的流式图计算模型，然而此模型对于无法满足这三个特征的算法不适用，因此模型还有很大的提升空间，可以再充分考虑其它图算法，采用归纳总结的方式抽象出它们的基本特征，进而丰富和完善本文所提出的模型。
- 算法方向：本文只将图算法中典型的四个图算法改造成了流式图算法，而对于其它的图算法没有继续进行改造，但是本文提供了详细的算法设计步骤和设计方法，因此读者可以根据本文提出的基于状态更新传播的流式图计算模型来设计和实现更多的动态图算法。

参考文献

[1] https://www.statista.com/topics/751/facebook/

[2] Malewicz G, Austern M H, Bik A J C, et al. Pregel: a system for large-scale graph processing[C]//Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010: 135-146.

[3] Xin R S, Gonzalez J E, Franklin M J, et al. Graphx: A resilient distributed graph system on spark[C]//First International Workshop on Graph Data Management Experiences and Systems. ACM, 2013: 2.

[4] Valiant L G. A bridging model for parallel computation[J]. Communications of the ACM, 1990, 33(8): 103-111.

[5] 申林,薛继龙,曲直,杨智,代亚非. IncGraph:支持实时计算的大规模增量图处理系统[J]. 计算机科学与探索,2013,12:1083-1092.

[6] Bar-Yossef Z, Kumar R, Sivakumar D. Reductions in streaming algorithms, with an application to counting triangles in graphs[C]//Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, 2002: 623-632.

[7] Tsourakakis C E, Kang U, Miller G L, et al. Doulion: counting triangles in massive graphs with a coin[C]//Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2009: 837-846.

[8] Buriol L S, Frahling G, Leonardi S, et al. Counting triangles in data streams[C]//Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. ACM, 2006: 253-262.

[9] S. Baswana. Streaming algorithm for graph spanners – single pass and constant processing time per edge. Inf. Process. Lett. 106(3):110–114, 2008.

[10] M. Elkin. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. ACM Transactions on Algorithms, 7(2):20, 2011.

[11] A. A. Bencz′ur and D. R. Karger. Approximating s-t minimum cuts in ˜O(n2) time. In ACM Symposium on Theory of Computing, pages 47–55, 1996.

[12] Chu S, Cheng J. Triangle listing in massive networks and its applications[C]//Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2011: 672-680.

[13] Cheng R, Hong J, Kyrola A, et al. Kineograph: taking the pulse of a fast-changing and connected world[C]//Proceedings of the 7th ACM european conference on Computer Systems. ACM, 2012: 85-98.

[14] 景年强,薛继龙,曲直,杨智,代亚非. SpecGraph:基于并发更新的分布式实时图计算模型[J].

计算机研究与发展,2014,(S1):155-160.

[15] Adamic L A, Lukose R M, Puniyani A R, et al. Search in power-law networks[J]. Physical review E, 2001, 64(4): 046135.

[16] Doekemeijer N, Varbanescu A L. A survey of parallel graph processing frameworks[J]. Delft University of Technology, 2014.

[17] Tian Y, Balmin A, Corsten S A, et al. From think like a vertex to think like a graph[J]. Proceedings of the VLDB Endowment, 2013, 7(3): 193-204.

[18] Malicevic J. Trends in Large-Scale Graph Processing[J].

[19] Kalavri V, Vlassov V, Haridi S. High-Level Programming Abstractions for Distributed Graph Processing[J]. arXiv preprint arXiv:1607.02646, 2016.

[20] Buluç A, Meyerhenke H, Safro I, et al. Recent advances in graph partitioning[M]//Algorithm Engineering. Springer International Publishing, 2016: 117-158.

[21] Rahimian F, Payberah A H, Girdzijauskas S, et al. Distributed vertex-cut partitioning[C]//IFIP International Conference on Distributed Applications and Interoperable Systems. Springer Berlin Heidelberg, 2014: 186-200.

[22] Buluç A, Meyerhenke H, Safro I, et al. Recent advances in graph partitioning[M]//Algorithm Engineering. Springer International Publishing, 2016: 117-158.

[23] Liu W, Wang J, Kumar S, et al. Hashing with graphs[C]//Proceedings of the 28th international conference on machine learning (ICML-11). 2011: 1-8.

[24] Ugander J, Backstrom L. Balanced label propagation for partitioning massive graphs[C]//Proceedings of the sixth ACM international conference on Web search and data mining. ACM, 2013: 507-516.

[25] Kernighan B W, Lin S. An efficient heuristic procedure for partitioning graphs[J]. The Bell system technical journal, 1970, 49(2): 291-307.

[26] Karypis G, Kumar V. METIS--unstructured graph partitioning and sparse matrix ordering system, version 2.0[J]. 1995.

[27] Ioanna Filippidou and Yannis Kotidis. Online and On-demand Partitioning of Streaming Graphs. In 2015 IEEE International Conference on Big Data (Big Data), 2015.

[28] Isabelle Stanton and Gabriel Kliot. Streaming Graph Partitioning for Large Distributed Graphs. In KDD '12, pages 1222-1230, 2012.

[29] Charalampos E. Tsourakakis, Christos Gkantsidis, Bozidar Radunovic and Milan Vojnovic. FENNEL Streaming Graph Partitioning for Massive Scale Graphs. In WSDM '14, pages 333-342. 2014

[30] Xie C, Chen R, Guan H, et al. Sync or async: Time to fuse for distributed graph-parallel computation[J]. ACM SIGPLAN Notices, 2015, 50(8): 194-204.

[31] Low Y, Gonzalez J E, Kyrola A, et al. Graphlab: A new framework for parallel machine

learning[J]. arXiv preprint arXiv:1408.2041, 2014.

[32] Lumsdaine A, Gregor D, Hendrickson B, et al. Challenges in parallel graph processing[J]. Parallel Processing Letters, 2007, 17(01): 5-20.

[33] Gelly I. Graph Processing with Apache Flink[J]. 2016.

[34] Avery C. Giraph: Large-scale graph processing infrastructure on hadoop[J]. Proceedings of the Hadoop Summit. Santa Clara, 2011, 11.

[35] https://hama.apache.org/hama_graph_tutorial.html

[36] Gonzalez J E, Low Y, Gu H, et al. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs[C]//OSDI. 2012, 12(1): 2.

[37] Kyrola A, Blelloch G E, Guestrin C. GraphChi: Large-Scale Graph Computation on Just a PC[C]//OSDI. 2012, 12: 31-46.

[38] S. Muthukrishnan. Data Streams: Algorithms and Applications. Foundations and Trends in Theoretical Computer Science, 1(2), 2005.

[39] S. Baswana. Streaming algorithm for graph spanners – single pass and constant processing time per edge. *Inf. Process. Lett.* 106(3):110–114, 2008.

[40] M. Elkin. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. *ACM Transactions on Algorithms*, 7(2):20, 2011.

[41] A. A. Bencz′ur and D. R. Karger. Approximating *s-t* minimum cuts in ˜*O*(*n*2) time. In *ACM Symposium on Theory of Computing*, pages 47–55, 1996.

[42] D. A. Spielman and S.-H. Teng. Spectral sparsification of graphs. *SIAM J. Comput.*, 40(4):981–1025, 2011.

[43] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. J. Algorithms, 55(1):58–75, 2005.

[44] P. Zhao, C. C. Aggarwal, and M. Wang. gSketch: On query estimation in graph streams. PVLDB, 5(3):193–204, 2011.

[45] Nan Tang, Qing Chen, Prasenjit Mitra. Graph Stream Summarization: From Big Bang to Big Crunch. SIGMOD '16 Proceedings of the 2016 International Conference. pages 1481-1496, 2016

[46] Sundaresh R S, Hudak P. A theory of incremental computation and its application[C]//Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1991: 1-13.

[47] Peng D, Dabek F. Large-scale Incremental Processing Using Distributed Transactions and Notifications[C]//OSDI. 2010, 10: 1-15.

[48] 张钟. 大规模图上的最短路径问题研究[D].中国科学技术大学,2014.

[49] Page L, Brin S, Motwani R, et al. The PageRank citation ranking: bringing order to the web[J].

[50] https://hazelcast.org/

[51] http://www.livejournal.com/

发表文章目录

段世凯等，专利《一种基于混合存储的流式数据自适应持久化方法及系统》，2016年

致谢

时光荏苒，青春不再。学生时代即将结束，感谢在这最后的时光里遇见了中国科学院软件研究所软工中心实验室。在这个温暖和谐、团结向上的大家庭中，我的知识储备得以丰富，综合素质得以提高，这为以后的工作生涯打下了坚实的基础。

首先，要感谢我的导师王伟副研究员。王伟老师是一位学识渊博、待人温厚、德才兼修的青年才俊，不仅在分布式存储和计算领域有很深的造诣，而且在实际的项目实践中也经验丰富，总能够在我茫然若迷时给我指点迷津，在生活中王老师更是细心体贴，嘘寒问暖，像哥哥一样照顾组里的每位成员，能够遇到这样一位知识渊博而又平易近人的好老师，真是倍感幸福！

我还要感谢我的师兄许利杰博士。许利杰师兄做事认真、态度严谨，在机器学习领域研究深入，对现有的分布式计算框架都了然于胸。许师兄凭借丰富的分布式计算知识储备，为我在论文的创作中提供了不少思路，凭借深厚的科研功底，为我今后思考和解决问题提供了宝贵经验。虽与许师兄共事时间不长，但其个人魅力对我影响深远！

我还要感谢同组的师兄唐震、支孟轩、任仲山和舒扬，同级的郑莹莹、刘财政，师妹沈雯婷、师弟赵伟和刘重瑞，以及同寝室的崔光范和倪嘉志，衷心的感谢你们，在这相处的时光里，无论是工作上还是生活中，你们都给予我很多帮助和关心，也正是这样一群团结友爱的小伙伴，才让科研生活如此的绚烂多彩！

我还要感谢软工中心实验室的全体老师和同学，感谢中国科学院软件研究所的全体老师和同学，在这三年的时光里，实验室和所里提供了如此完善的学习环境和生活环境，还经常举办很多有意思的课外活动和科研讲座，为我们的科研生活增添不少乐趣！

我还要感谢我的女朋友韩婷，她在生活中无微不至的照顾我，在科研中经常鼓励我，开导我，而且在最后的论文撰写和修改过程中也为我提供很多帮助，谢谢韩婷，让我在远离家乡上千里的北京有了家的温暖！

当然更要感谢我的父母和两位姐姐和姐夫，父母已过半百，靠在农村的辛苦劳作得来的微薄收入，供我和二姐上大学、读研究生，实在不易。谁言寸草心，报得三春晖，我相信一定会让父母过上舒适惬意的生活！

最后，衷心的感谢人生路上所有帮助过、教导过、批评过、鼓励过我的同学、朋友、老师和亲人，是你们让我感受知识的魅力，体会人生的乐趣！