# 摘要

# Abstract

# 目录

[摘要 1](#_Toc478234110)

[Abstract 1](#_Toc478234111)

[目录 1](#_Toc478234112)

[1. 绪论 5](#_Toc478234113)

[1.1. 研究背景 5](#_Toc478234114)

[1.2. 论文工作 6](#_Toc478234115)

[1.3. 论文组织 6](#_Toc478234116)

[2. 图计算相关工作 7](#_Toc478234117)

[2.1. 问题定义 7](#_Toc478234118)

[2.2. 关键技术 8](#_Toc478234119)

[2.2.1. 图的划分 9](#_Toc478234120)

[2.2.2. 编程模型 10](#_Toc478234121)

[2.2.3. 计算模型 10](#_Toc478234122)

[2.3. 常见图计算模型和系统 11](#_Toc478234123)

[2.3.1. 批处理图计算 11](#_Toc478234124)

[2.3.2. 流处理图计算 13](#_Toc478234125)

[2.3.3. 常见图系统总结 17](#_Toc478234126)

[2.4. 存在问题 17](#_Toc478234127)

[2.5. 本章小结 18](#_Toc478234128)

[3. 基于状态更新的流式图计算模型 19](#_Toc478234129)

[3.1. 流式场景下的图算法特征分析 19](#_Toc478234130)

[3.1.1. Degree Distribution 19](#_Toc478234131)

[3.1.2. Triangle Count 20](#_Toc478234132)

[3.1.3. Single Source Shortest Path 20](#_Toc478234133)

[3.1.4. PageRank 21](#_Toc478234134)

[3.1.5. 算法特征总结 22](#_Toc478234135)

[3.2. 模型定义 24](#_Toc478234136)

[3.2.1. 状态（State） 25](#_Toc478234137)

[3.2.2. 事件（Event） 25](#_Toc478234138)

[3.2.3. 转换（Transform） 26](#_Toc478234139)

[3.3. 状态存储和更新 26](#_Toc478234140)

[3.3.1. 状态类型 26](#_Toc478234141)

[3.3.2. 独立状态的存储和更新 27](#_Toc478234142)

[3.3.3. 关联状态的存储和更新 28](#_Toc478234143)

[3.4. 模型应用举例 30](#_Toc478234144)

[3.5. 本章小结 31](#_Toc478234145)

[4. 流式图算法设计 32](#_Toc478234146)

[4.1. Degree Distribution 32](#_Toc478234147)

[4.2. Triangle Count 32](#_Toc478234148)

[4.3. Single Source Shortest Path 34](#_Toc478234149)

[4.4. PageRank 38](#_Toc478234150)

[4.5. 本章小结 41](#_Toc478234151)

[5. GraphFlow系统的设计与实现 42](#_Toc478234152)

[5.1. 系统架构 42](#_Toc478234153)

[5.2. 模型实现 44](#_Toc478234154)

[5.2.1. 状态（State） 44](#_Toc478234155)

[5.2.2. 事件（Event） 46](#_Toc478234156)

[5.2.3. 转换（Transform） 48](#_Toc478234157)

[5.3. 算法实现 48](#_Toc478234158)

[5.3.1. Triangle Count 49](#_Toc478234159)

[5.3.2. Single Source Shortest Path 50](#_Toc478234160)

[5.4. 本章小结 51](#_Toc478234161)

[6. 实验与分析 52](#_Toc478234162)

[6.1. 实验环境 52](#_Toc478234163)

[6.2. 实验结果 53](#_Toc478234164)

[6.2.1. 准确性 53](#_Toc478234165)

[6.2.2. 实时性 54](#_Toc478234166)

[6.2.3. 更新冲突概率 56](#_Toc478234167)

[6.3. 本章小结 56](#_Toc478234168)

[7. 结束语 57](#_Toc478234169)

[7.1. 工作总结 57](#_Toc478234170)

[7.2. 下一步工作 58](#_Toc478234171)

[参考文献 59](#_Toc478234172)

[发表文章目录 59](#_Toc478234173)

[作者简历 59](#_Toc478234174)

[致谢 59](#_Toc478234175)

# 绪论

## 研究背景

图是计算机科学中常用的一类数据结构，它很好的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的表达数据之间的关联性和聚集情况，因此针对图数据表达的关联关系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，就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的处理图数据，也使得计算延迟显著增加；另一方面，这些数据又是动态变化，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分布式的集群和BSP（Bulk Synchronized Parallel）[4]消息同步机制来处理图数据。然而这些系统都是在静态的图数据结构上进行的离线批量处理,即每次针对整体的图进行计算，当图动态变化时，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，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的方式来降低时间和空间开销，并通过特定的采样方法来减少真实值和实测值之间的差距。 如文献[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；文献[9][10][11]通过将原始的图信息转化为简单的数据结构来保存图中元素，降低了内存消耗。虽然估计算法能够在一定程度上节约了内存和计算开销，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]提出采用增量计算的模型进行实时计算，然而这种增量式的更新是串行执行的，实时性有限。SpecGraph[14]虽然在上述两个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，然而该模型中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。

## 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**基于状态更新的动态图计算模型**，能够在原有图状态上并发的计算增量信息对状态的影响，而无需在整个图上重新计算，同时通过控制更新影响范围来提高并发性，实现状态的并发更新。本文的工作主要有以下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、编程模型和计算模型三个方讲解如何针对图数据和图计算的特点提出的关键技术，然后按照图数据的处理方式不同，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）给出了图算法的特征分析方法。从四个典型的图算法--节点度分布（DD, Degree Distribution），三角形数目（TC, Triangle Count），单源点最短路径（SSSP, Single Source Shortest Path），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他图算法。

（3）根据流式图算法的特点，建立了基于状态更新的流式图计算模型。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，完成了图状态的转变，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有效提高系统的实时性和准确性。

（4）给出了在基于状态更新的流式图计算模型上设计算法的思路和步骤。并以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他符合条件的批式图算法改成流式图算法。

（5）从正确性和实时性对系统实现进行评估。结果表明：基于状态更新的动态图计算模型构建的算法能够得到较为正确的计算结果，计算偏差在5%以内；而且90%的更新请求都能够在12ms内得到响应，符合实时性要求。

## 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，并且从图的划分、编程模型和计算模型三个层面分析了大规模图计算所需的关键技术，最后对常见的图计算的系统进行了比较。
* 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的动态图计算模型，来对其他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、模型实现和算法实现三个层面，详细阐述GraphFlow系统的设计和实现细节。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了一个总结，并且对以后改进的方向提出了几点展望。

# 图计算相关工作

## 问题定义

图是计算机中一种常用的数据结构，常常用来表达个体之间的关联性。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，为其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记录为她们推荐商品；在万维网中，网页以及网页之间的相互链接可以形成巨型网页图，根据链接信息来对网页的重要性进行排名；由此可见，图数据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”两个特性。“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。图结构很好的表述了实体之间的关联，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，只有极少数的节点的度很高，大部分节点的度都非常低。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，导致节点之间的通信代价提高。此外，图相关的算法往往不是一次计算就可以获得最终结果，而是需要通过多次迭代计算，达到收敛条件或者最大迭代次数时算法才运行结束。



图 2‑2 满足Power-Law规则的图的节点的度分布图

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

针对图数据的海量和动态特性，以及图算法的局部性差和迭代计算的特点，下面我们列举并分析了现有的图计算的关键技术点。

## 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。在本节中我们重点选取了底层细节和模型两个角度来阐述，而在算法和系统层面，我们会在其他章节继续讨论。



图 2‑3 图计算技术概览

从底层细节角度来看，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数据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进行计算。因此图的划分在图计算中占据重要地位：一方面这种划分要保证负载均衡，即希望能够将整个图数据均匀的划分到各个计算节点上；另一方面又希望这种划分能够较好的维持图数据的关联性，即相互关联的节点希望它们能够分配到同一个计算节点上，以减少节点之间的通信。

从模型角度来看，图计算模型可以分解为编程模型和计算模型：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、SG、GSA都是以顶点为中心的编程模型等；而计算模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点来详细阐述图计算中的关键技术。

### 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的图数据均衡的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变得复杂。其一考虑到节点负载的均衡性，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，又希望相互邻接点能够划分到一起来减少不同计算节点之间的通信代价。如果划分的不好，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。

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，然后将这两个顶点分配到不同的计算节点上，如图2-3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，通过将中心节点切割，将该节点的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。

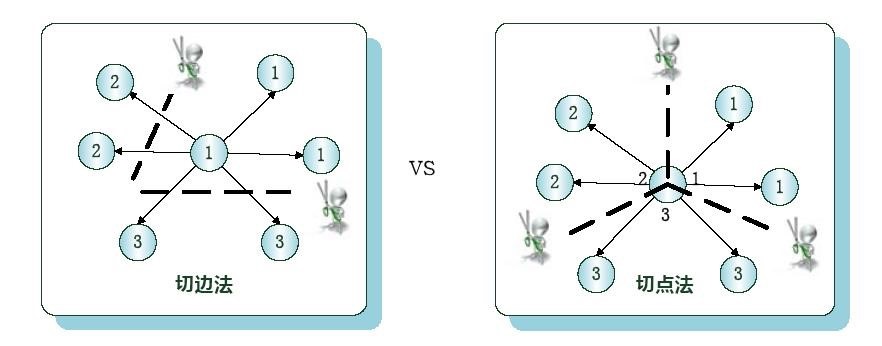


图 2‑4 切边法和切点法

### 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[]。考虑到最为常用的是以顶点为中心的编程模型，下面我们将重点介绍Vertex-Centric及其扩展Gather-Apply-Scatter。

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的将这种变化传播给其他顶点。顶点为中心的计算模型具有很强的表达能力，很多图算法都可以用该模型来进行表示，也是图计算中最为常用的一种编程模型。

Gather-Apply-Scatter编程模型可以看成是Vertex-Centric编程模型的细粒度的改造。他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，节点收集其他节点（一般是邻接点或者副本节点）发送过来的信息，在Apply阶段，节点根据Gather阶段收集的信息和节点原来的状态对节点的当前状态进行更新，在Scatter阶段，节点会根据Apply阶段的更新情况，决定更新信息传播给哪些节点。这种细粒度的划分，使得系统能够更大程度的提高不同算子的并发度。在后文的PowerGraph系统中，我们会详细阐述该模型是如何工作的。

### 计算模型

图的计算模型是指图的底层执行引擎以何种执行方式来完成图计算。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，节点的变化能够立即被其他节点看到并使用，则称这种计算模式为异步计算模型，相反如果所有节点的变化只有在下一轮的迭代过程中才能被其他节点看见和使用，则称这种计算模式为同步计算模型。

同步计算模型可以用图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，同步点之间的称为一个迭代阶段，在每个迭代阶段内部，各个并发程序之间可以并发的执行计算，在每个同步点上，各个并发程序之间交换彼此计算结果信息。



图 2‑5 同步计算模型

BSP模型和MapReduce模型都是典型的同步计算模型。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。在2.3节我们会详细阐述BSP模型在图计算领域中的运用。

异步计算模型可以用图2-6来描述。相比较同步计算模型，异步计算模型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，但程序的正确性无法获得保证。



图 2‑6 异步计算模型

## 常见图计算模型和系统

现在有诸如Pregel，Spark GraphX，GraphLab等很多成熟的图计算框架和系统，本文按照图数据的特性，将这些系统分为批处理图计算系统和流处理图计算系统，即批处理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；而流处理图计算系统处理的图数据是动态的，针对动态变化的图数据，能够及时反馈中间计算结果，适合在线图计算情景。

### 批处理图计算

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为代表的传统面向数据并行（Data-Parallel）的计算模型难以对图计算提供高效的支持。[17]

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。[18]专门解决网页链接分析、社交数据挖掘等图计算问题。Pregel使用了以顶点为中心的计算模型,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，活跃的节点(active vertex)将将接收上个超步中其他节点发送过来的消息，并执行用户自定义的计算函数，改变自己的状态，同时将更新的状态再发送给其他节点，这些消息会在下个超步中被其他节点接收并处理，然后该点进入不活跃状态（inactive vertex）。不活跃的点在下个超步中接收到其他节点的消息会变得活跃，反之如果没有接收其他节点的消息，将继续保持不活跃的状态，也不会向其他节点发送消息。超步内各个节点可以并行处理，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，直至所有节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节点的计算逻辑，即可实现计算功能。继Pregel之后，一大批以BSP为模型的分布式图计算系统涌现，如Spark GraphX，Flink Gelly，Giraph，Hama等，他们都是借助于同步计算模型和Vertex-Centric编程模型，实现了SSSP和PR等图算法。



图 2‑7 BSP模型迭代计算过程

鉴于现有的BSP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模型的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，对图模型表示的数据进行切分[]，对Vertex-Centric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。

考虑到现实世界中的自然图一般符合Power Low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，所以一个节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节点为主节点，其他机器上的节点为镜像节点，在Gather阶段，各个镜像节点将自身累积的信息发送给主节点，主节点Apply阶段更新其值之后，将最新的信息通知给其他镜像节点。在接下来的Scatter阶段，各个镜像节点可以并发执行，去更改邻接点或邻接边的值。其各个阶段的执行过程如下：

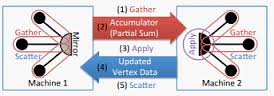


图 2‑8 PowerGraph GAS执行过程图

除了上述同步和异步计算模型的分布式并行图计算系统能够处理静态图数据外，GrapLab实验室推出的单机版的GraphChi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，因此不做过多介绍。

### 流处理图计算

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，流处理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处理模型和计算模型，最后介绍了支持增量计算的分布式准实时流式图计算系统KineoGraph。

#### 流图处理模型

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）不再是静态的存储在文件或数据库中，而是以流的形式源源不断的添加到系统中。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模型[7]：

* 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，我们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2-7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，本文考虑的是边的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

#### 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的数据结构的方式，不存或者存储少量的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，采用该模型的典型算法为triangle count[5]，定义了两类图流arbitrary stream和incidence stream，并分别提出了1-pass和3-pass采样算法。
* 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) [8, 9]: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) [10, 11]: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)[12, 13]和同构草图(homomorphic sketch)[14]，，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了内存和计算开销，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，学术界很早就开始对大数据领域中增量计算进行了研究。如Google Percolator系统在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2-8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，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。

#### KineoGraph系统

KineoGraph是一个支持增量计算的分布式准实时的流式图计算系统。它采用Vertex-Centric编程模型，以增量的方式串行执行更新操作。KineoGraph的系统架构图如图2-8所示：

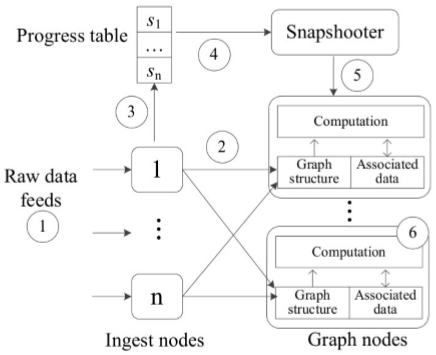


图 2‑11 KineoGraph系统架构图

各个组件的功能如下：

1. 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；
2. 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；
3. 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；
4. 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．[5]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。

### 常见图系统总结

现有常见的图计算系统如表格2-1所示，通过分析该表，我们可以得出以下结论：（1）针对批处理的图系统，除了GraphLab这类专门处理符合Power Low规则的专用系统外，大部分系统都是采用了节点随机哈希（本质上是切边法）的划分原则，Vertex-Centric的编程模型和BSP计算模型，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，性能表现更加出色；（2）针对流处理图计算系统，一般也是采用Vertex-Centric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，至于像Pregel这种商业界和学术界都认可的批处理图计算系统建设少之又少，因此存在很大的发展空间和研究价值。

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 |
| 批处理图计算系统  适合离线计算 | Spark GraphX | 节点随机哈希 | Vertex-Centric | BSP |
| Flink Gelly | 节点随机哈希 | Vertex-Centric | BSP |
| Pregel | 节点随机哈希 | Vertex-Centric | BSP |
| GraphLab | 节点随机哈希 | GAS | 异步执行 |
| PowerGraph | 节点随机哈希  Edge-Cut  Vertex-Cut  启发式v-cut | GAS | 混合 |
| Giraph | 节点随机哈希 | Vertex-Centric | BSP |
| 流处理图计算系统  支持增量计算 | KineoGraph | 节点随机哈希 | Vertex-Centric | 增量计算 |
| SpecGraph | 未明确表示 | Vertex-Centric | 增量计算 |

## 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，我们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**研究深度**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[][][]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，缺乏对整体的图计算算法的特征归纳总结，也就无法建立更为通用的计算模型和编程模型。而且现有的流式图计算系统也存在各种缺陷，诸如KineoGraph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，但其推出的解耦合计算模型是建立在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的假设基础上。而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，抽取图算法的典型特征，然后在这些特征的基础上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图算法，以验证模型的正确性。

## 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、图的编程模型和图的计算模型这三个角度，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和本文的解决思路。

# 基于状态更新的流式图计算模型

在本节中，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的动态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个算法说明如何在该模型上设计动态图算法。

## 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，我们选取了DD、TC、SSSP、和PR四个算法来进行分析,我们约定，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。

### Degree Distribution

DD算法是统计无向图中各个节点的度，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节点的度。图中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。

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。

1. T1时刻 (b)T2时刻

图 3‑1 无向图中各个节点的度

### Triangle Count

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节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

### Single Source Shortest Path

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。[16]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节点之间的距离，顶点内的数字表示当前时刻源点到该节点的最短距离，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两个顶点，甚至会以这个两个点为中心，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（即在顶点原有SP值的基础上取最小值），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1顶点影响，也可能会受白色的SP值为3的顶点影响。

1. T1时刻 (b)T2时刻

图 3‑3 流式场景下的SSSP算法

### PageRank

PR算法[17]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（即PageRank值，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节点的PR值。

在T1时刻，各个节点的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,原来的PR值为,新增一条边后出度变为+1,那么它对于邻接点的贡献由原来的变化为，进而影响了这两个顶点的所有邻接点的PR值，如图3-4(b)中灰色节点所示，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节点的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节点。在顶点PR值更新过程中，是利用原有的顶点状态进行更新(如新增变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。

1. T1时刻 (b)T2时刻

图 3‑4 流式场景下的PR算法

### 算法特征总结

当新增一条边时，本文从影响范围、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节点；（2）计算方法是指采用何种计算模型来进行计算；（3）计算顺序是指被影响的节点谁先参与计算对最终的计算结果是否相关；（4）计算特性是指被影响的节点的更新函数满足哪些代数运算的性质；（5）计算次数是指这种影响是否会被计算多次，例如在DD算法中，新增加一条边只需要将这条边的源点和目标点对应的度数加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，这条边的源点和目标点的输出贡献将会发生变化，因此会首先影响它们的所有邻接点，这些邻接点在下一次的传播中会继续它们的邻接点，经过多次迭代计算之后各个节点的PR值会趋于稳定，在迭代计算的过程中，每个节点可能参与多次计算。

需要注意的是，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点和目标点 | 利用原始状态进行增量式计算 | 最终计算结果和被影响的节点的计算顺序无关 | 更新函数为加法运算 | 被影响的节点只参与计算一次 |
| **TC** | 影响新增这条边的源点和目标点，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节点的计算顺序无关 | 更新函数为加法运算 | 被影响的节点只参与计算一次 |
| **SSSP** | 以这条边的某个节点为起点，沿着某条路径往其他节点传播影响 | 利用原始状态进行增量式计算 | 最终计算结果和被影响的节点的计算顺序无关 | 更新函数为Min运算 | 被影响的节点可能会参与计算多次 |
| **PR** | 影响这条边的源点和目标点所在的整个连通子图内的所有节点 | 利用原始状态进行增量式计算 | 最终计算结果和被影响的节点的计算顺序无关 | 更新函数为累加运算 | 被影响的节点一般会参与计算多次 |

通过分析这四类的典型图算法，我们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，我们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，而其他节点的度都保持不变，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？如果被影响的节点的更新顺序与最终的计算结果无关，我们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，因此被影响的节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，他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，而该函数满足:① = ，②,即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：

（公式1）

为当前时刻图的所有顶点的集合，为图的顶点数目，为指向顶点的所有顶点集合，为顶点指向其他顶点的集合，为调整因子，使得系统可以按照一定的概率跳转向图中其他任意节点，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，考虑到公式中和均为常量，因此我们只分析公式中的。设指向顶点的顶点为,则原式可化简为：

对于个元素的求和函数，也是满足交换律和结合律的。因此，这四个算法，在流式场景下的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，我们可以实现面向流式图数据的算法。为此，本文首先建立了基于状态更新的流式图计算模型，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。

## 模型定义

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的流式图计算模型，能够很好的解决图不断变化时的计算问题。基于状态更新的流式图计算模型，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 基于状态更新的流式图计算模型

基于状态更新的流式图计算模型有如下几个定义：

### 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。

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。

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 |

### 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 |

### 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。

## 状态存储和更新

### 状态类型

基于状态更新的动态图计算模型中，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，也可以以边或者其他的方式来组织。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他因子的状态的变化，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节点，不会影响到其他的节点；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，而在影响返回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

### 独立状态的存储和更新

对于独立状态，因为状态内的各个因子之间不会相互影响，因此独立状态可以分布式的存储和并发的更新。即可以按照状态的组织形式，将图的状态分布式的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。独立状态的并发更新过程如图8所示：



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存储和更新的过程。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。

### 关联状态的存储和更新

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，其他任何事件都不得触发更新，以免引起更新冲突问题。诸如IncGraph[5]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，使得子图内部的节点之间联系比较紧密，子图之间的节点之间几乎没有边相连或者联系较少。这样可以假设子图内节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节点。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。如图9所示。



图 9 基于分区的并行更新策略

图9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如图9所示的四个连通子图，在每个连通子图的更新是串行的，如在左上角子图中a节点和f节点的更新需串行进行，而连通子图之间的节点的更新是并行的，如a节点、e节点、k节点和f节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的选择子图划分算法，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，保证子图之间节点的联系是松散的，子图内部的节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[18]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[19]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[20]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，平衡分割时间更短，效果更好。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的会出现多个节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。所以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节点数据。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节点。如图3-8所示。该锁以因子为单位，每次因子更新请求时，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他顶点。



图 3‑8 基于细粒度锁的因子更新过程图

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，超步内部的节点之间并行计算，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，而且没有显示的同步过程，消除了短板效应。但这需要因子的更新满足（1）序列一致性；（2）交换律和结合律，即对于任何一个节点的若干个更新，这些更新的顺序不会影响整个节点的最终状态，而且因子的更新函数符合代数运算的交换律和结合律。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，本文在6.2.3实验证明了这种冲突的概率不会超过5%，因此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。

## 模型应用举例

基于状态更新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，基于状态更新的动态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，，其中表示第个连通分支，, 其中集合表示由这些顶点构成了一个连通分支，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，其中，表示新增边 ；

（3）Transform:图在动态变化过程中，State的更新过程见算法1：

|  |
| --- |
| **算法 1 Dynamic Connected Components** |
| 1 ***for*** each  ∈ EventStream  2 ***do* (,)** ← GET-VALUE()  3 ← GET-STATE()  4 ← GET-STATE()  5 ***if*** ≠ ∅ ***and*** ≠ ∅  6 ***then*** UNION-STATE(,)  7 ***elseif*** ≠ ∅ ***and*** = ∅  8 ***then*** UNION-STATE(,)  9 ***elseif*** = ∅ ***and*** ≠ ∅  10 ***then*** UNION-STATE(,)  11 ***else***  12 s ← (min(**,**),{**,**})  13 ADD-STATE(s) |

在算法1中，表示节点所在的连通分支，表示节点所在的连通分支，新增的这条边**,**使得这两个节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。

## 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；在分析完成后，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算法举例说明如何利用该模型进行算法设计。

# 流式图算法设计

在第3章节，我们介绍了基于状态更新的流式图计算模型，并且分析了流式场景下图算法的特点，在本章节，我们使用该模型提供的组件，讲解DD、TC、SSSP和PR的流式算法如何进行设计。

## Degree Distribution

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

1. State: 当前图中各个顶点的度值，即，其中，表示顶点的度为；
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 ，其中，,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2所示：

|  |
| --- |
| **算法2 Dynamic Degree Distribution** |
| 1 ***for*** each Event Stream  2 ***do*** (,) ← GET-VALUE()  3 type GET-TYPE()  4 ← GET-STATE()  5 ← GET-STATE()  6 ***if*** type = ADD  7 ***then*** SET-STATE(,+1)  8 SET-STATE(+1)  9 ***elseif*** type = DELETE  10 ***then*** SET-STATE(,-1)  11 SET-STATE(-1) |

## Triangle Count

Triangle Count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， ，其中，表示顶点的邻接点的集合为， 顶点构成的三角形的数目为；

（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 ，其中，,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 Event Stream  2  ***do*** ← ∅  3 ← ∅  4  **(,)** ← GET-VALUE()  5 ← GET-STATE()  6 ←GET-STATE()  7 ***if*** ≠ ∅  8 ***then*** ← GET-NEIGHBOR() ∪ {}  9 ***else*** ← {}  10 ***if*** ≠ ∅  11 ***then*** ← GET-NEIGHBOR() ∪ {}  12 ***else*** ← {}  13 cross ←∩  14 type ← GET-TYPE()  16 ***for*** each ∈ cross  17 ***do*** ← GET-STATE()  18 t ← GET-TRIANGLE()  19 N ← GET-NEIGHBOR()  20 ***if*** type = ADD ***then*** ←(,N,t+1)  21 ***elseif*** type = DELETE ***then*** ←(,N,t-1)  22 SET-STATE(,)  23 ← GET-TRIANGLE()  24 ← GET-TRIANGLE()  25 ***if*** type = ADD  26 ***then*** ← (**,**,+|cross|)  27 ← (**,** , +|cross|)  28 ***elseif*** type = DELETE  29 ←(**,**,-|cross|)  30 ←(**,** ,-|cross|)  31 SET-STATE(**,**)  32 SET-STATE(**,**) |

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，用来存储顶点和的邻接点；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行用来更新顶点和的邻接点；第13行用来求解这两个集合的交集，即和的公共邻接点，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行用来更新顶点和的状态—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|。

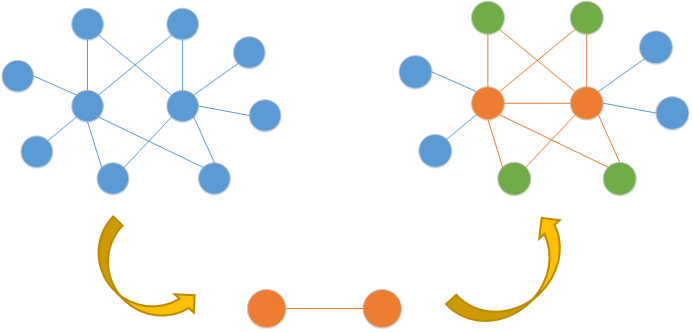


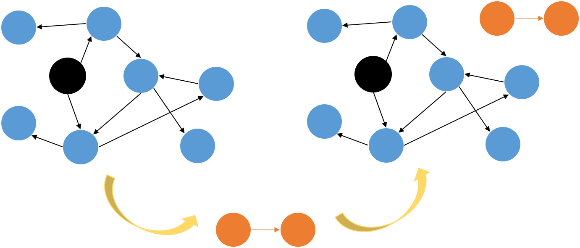
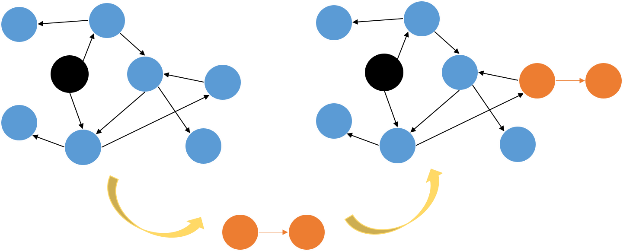
图 4‑1 Dynamic Triangle Count

如图4.1所示，左边蓝色的图为当前时刻图的状态，在增加一条边之后，即图中黄色顶点及其连线，这会使得整个图新增了4个三角形，这4个三角形即为新增边的两个顶点的公共邻接点。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|。

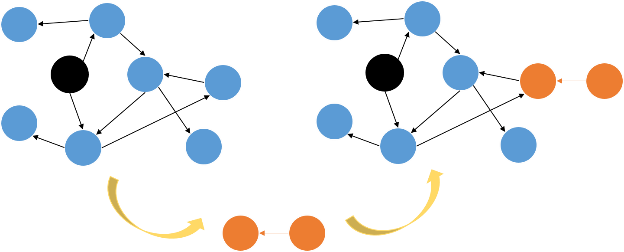
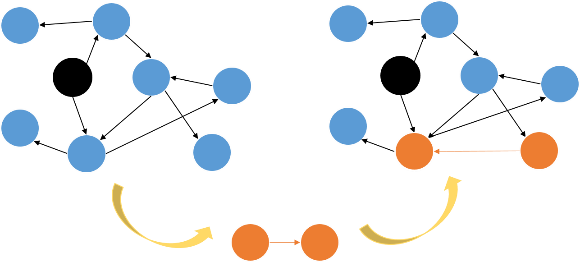
## Single Source Shortest Path

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，边的方向为指向。则一共有如图4-2所示的(a)均为新顶点；(b)为新顶点,已经存在于系统中；(c)为新顶点，已经存在于系统中；(d)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，橘色顶点为顶点，蓝色的顶点是图中其他已经存在的顶点。在每个子图中，左边部分是原图，中间橘色的顶点和连线是新增的边，右图为新增边之后的新图。

(a) 均为新顶点 (b)为新顶点,

(c)为新顶点，已经存在于系统中 (d)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）均为新顶点

如图4-2(a)所示，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）为新顶点，已经存在于系统中

如图4-2(b)所示，为原图中已经存在的顶点，为原图中不存在的新增加的顶点，且有指向，此时，因为指向的顶点集合没有变化，所以的SP值不会发生改变；而又有指向，因此可由到达，所以的SP值更新为,其中为的SP值，为边的长度。

（c）为新顶点，已经存在于系统中

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

（d）均已经存在于系统中

① ADD和UPDATE事件

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

(公式2)

即取的原来的SP值和从过来到达的值的最小值。如果的SP值变小，则的后续顶点的SP值可能因为的变小而变小，因此当的SP值变小时，需要将这种变化传播给所有指向的邻接点，同时，这些邻接点又可能继续将这种影响传播出去；而当的值没有发生改变时，说明从过来的路径不是最短路径，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

如果从左往右看图4-2(d)，则是新增边的过程，如果从右往左看图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的SP值。在原图中，当时，说明的最短路径并不是从过来的，即删除这条边并不会影响的SP值，而且也不会影响的SP值；当时，并不是说的最短路径一定是从过来，因为还有可能存在其他的顶点到达的路径和相同，因此需要判断是到达的唯一的最短路径，那么此时删掉，一定会引起的SP值发生改变，那么更新的SP值，再将这种改变传播给邻接点。

在分析完流式场景下的算法细节之后，下面我们将套用基于状态更新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， ，其中，表示顶点到源点的最短路径为；

（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 ，其中，,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 Event Stream  2  ***do*** **(**,,**)** ← GET-VALUE()  3 type ← GET-TYPE()  4 ← GET-STATE()  5 ←GET-STATE()  6 ← GET-STATE()  7 ←GET-STATE()  8 ***if*** = ∅ ***and*** = ∅  9 ***then*** SET-STATE(,+)  10 SET-STATE(,+)  11 ***elseif*** ≠ ∅ ***and*** = ∅  12 ***then*** SET-STATE(,)  13 ***elseif*** *=*∅ ***and*** ≠ ∅  14 ***then*** SET-STATE(,+)  15 ***else***  16  17 ***if*** type = ADD ***or*** type = UPDATE  18 ***if***  19 ***then*** SET-VALUE(,)  20 SPREAD-TO-OUT-NEIGHBOR()  21 ***elseif*** type = DELETE  22 ***if*** = ***and*** ONLY-SHORTEST-PATH(,)  23  ***then***  *=* FIND-SHORTEST-PATH()  24 SPREAD-TO-OUT-NEIGHBOR() |

算法2-7行是获取,顶点（为源顶点，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值加上边的长度；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，即新增或更新的这条边对图的状态没有任何影响；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是到达目标顶点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(,)即为判断是否是到的唯一最短路径，FIND-SHORTEST-PATH()计算除去顶点之后，到达顶点的最短路径的值。

## PageRank

PR算法[17]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（即PageRank值，简称PR值）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：

1. 均为新顶点

如图4-2(a)所示，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)：

源顶点的PR值：

目标顶点的PR值：

其中，是原图中的顶点数目。

1. 为新顶点,已经存在于系统中

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

1. 为新顶点，已经存在于系统中

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

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

此时顶点的PR值发生了变化，则以为中心将这种变化传播出去。

1. 均已经存在于系统中

① ADD事件

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

其中为调整因子，为指向的邻接点的集合，为指向的邻接点的集合的元素数目。

虽然的PR值不变，但是对其指向的邻接点的贡献发生了变化，所以需要以为中心，将这种变化传播到他们指向的邻接点。

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),此时对于顶点，指向的顶点集合没有发生改变，所以迭代开始时，的PR值不变；对于顶点，由于删除了指向的这条边，所以的PR值将在原来的基础上减少顶点对其的贡献，即

类似在① ADD事件的分析过程，虽然的PR值没有发生改变，但是由于删掉了指向的这条边，导致对其指向的其他邻接点的贡献发生了变化，因此需要以为中心，将这种变化传播到他们指向的邻接点。

在分析完流式场景下的算法细节后，下面我们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， ，其中，表示顶点的PR值为；

（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 ，其中，,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 Event Stream  2  ***do*** **(**,**)** ← GET-VALUE()  3 type ← GET-TYPE()  4 ← GET-STATE()  5 ←GET-STATE()  6 ← GET-STATE()  7 ←GET-STATE()  8 ***if*** = ∅ ***and*** = ∅  9 ***then*** SET-STATE(,)  10 SET-STATE(,)  11 ***elseif*** ≠ ∅ ***and*** = ∅  12 ***then*** SET-STATE(,)  13 SPREAD-TO-OUT-NEIGHBOR()  14 ***elseif*** *=*∅ ***and*** ≠ ∅  15 ***then*** SET-STATE(,)  16 SET-STATE(,)  17 SPREAD-TO-OUT-NEIGHBOR()  18 ***else***  19 ***if*** type = ADD ***then*** SET-VALUE(,)  20 ***elseif*** type = DELETE ***then*** SET-VALUE(,)  21 ***else return***  22 SPREAD-TO-OUT-NEIGHBOR()  23 SPREAD-TO-OUT-NEIGHBOR() |

算法5中，第2-7行是获取源顶点和目标顶点的PR值；8-10行对应（a）情况，当两个顶点都是新顶点时，设置他们的初始值分别为和；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继续传播影响；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行是以和为中心，向他们的邻接点传播这种影响。

SPREAD-TO-OUT-NEIGHBOR()函数是传播函数，它将触发向其指向的邻接点发送信息，这个信息就是对其指向的邻接点的贡献值，接收到所发信息的邻接点，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。

## 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，将图算法中其他算法设计成流式图算法。

# GraphFlow系统的设计与实现

## 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进行计算，使得系统的实时性强；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。

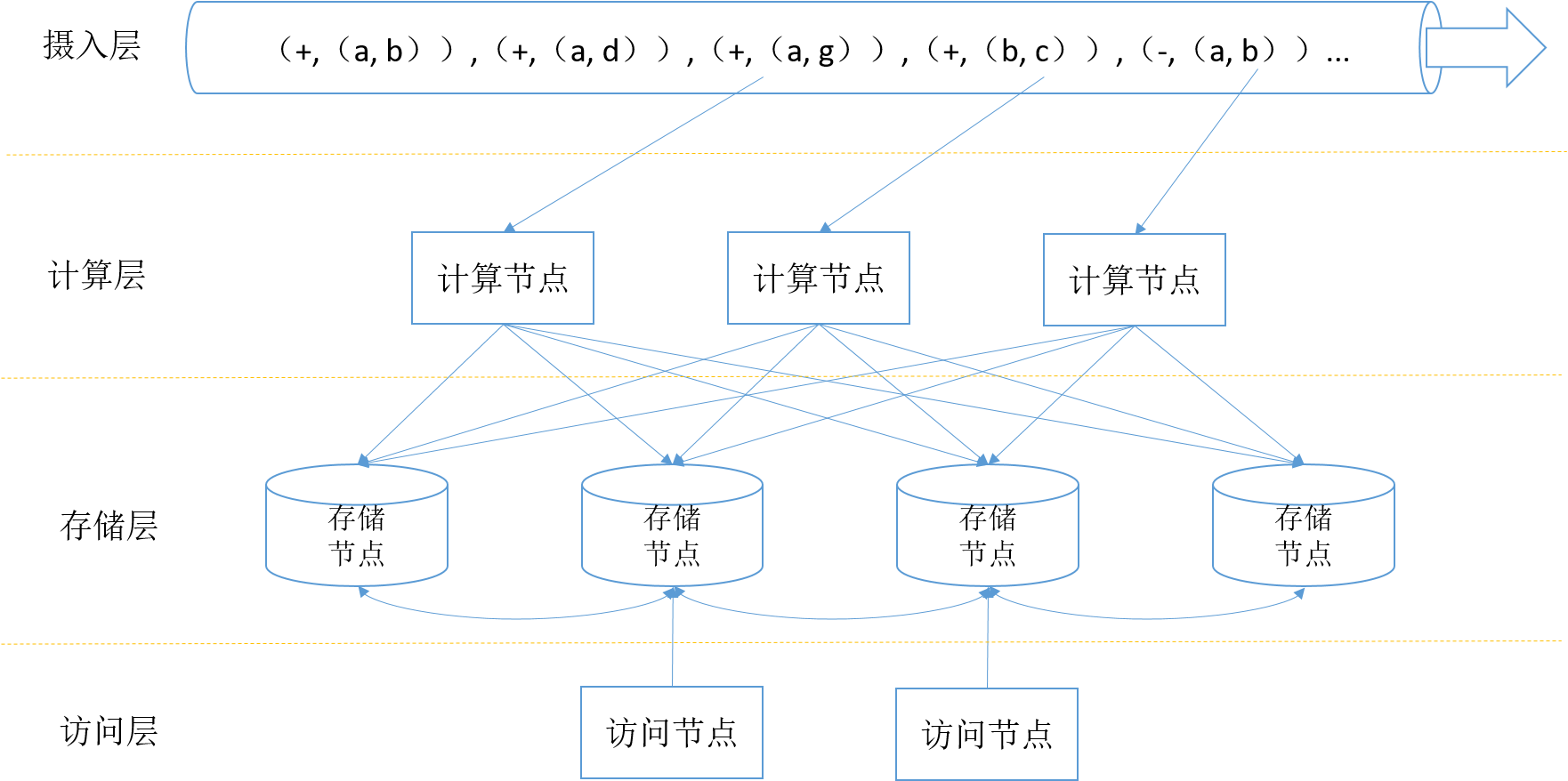


图 5‑1 GraphFlow系统架构图

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，也可以存在于如Kafka，HBase等其他分布式系统中。GraphFlow系统提供了接口，能够很好的将这些数据源集成到系统。
* 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，以便其他计算节点能够立刻感知到。
* 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计算结果。在本层我们利用开源产品内存数据网格Hazelcast来作为数据的存储层，存储图的状态信息。
* 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。在本层我们使用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，下面我们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新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他计算节点感知。该层又可以细分为：应用层、服务层、API层、核心层和存储层。



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。

## 模型实现

在第3章中，我们详细介绍了基于状态更新的流式图计算模型。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。下面我们将重点介绍这三个组件的实现。

### 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，本文将这种状态可以称之为离散状态（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）。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因子组成，用户可以通过set(KV,SV)设置因子的状态，也可以通过get(KV)来获取某个因子的状态。
* 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，我们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他的算法所使用，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，采用键值对的形式来存储数据，其中是顶点的标号，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，采用键值对的形式存储数据，其中是顶点标号，是所有指向该顶点的顶点集合，是该顶点指向其他顶点的顶点集合。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，采用键值对的形式来存储，其中是顶点的标号，是所指向的顶点，是所组成的边的值，是所有这样对的集合。相比较OutNeighborState类，她不仅记录了顶点所指向的邻接点标号，还记录了他们所组成的边的值。

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，我们在算法实现章节再做具体介绍，这里不再阐述。

### 事件（Event）

基于状态更新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。

### 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，我们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。

## 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。他们都继承自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自己特定的算法。

### Triangle Count



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。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。

|  |
| --- |
| **算法6 RUN()** |
| 1 publicvoidrun(){ 2 while(stream.hasNext()) { 3 EdgeEvent<KV, EV> edgeEvent = stream.next(); 4 state.update(edgeEvent); 5 } 6 } |

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的非常详细，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。

### Single Source Shortest Path



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。下面我们重点讲解一下在SSSP算法中，顶点的影响是如何传播给其他顶点的。

|  |
| --- |
| **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他们的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。

## 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，讲解了如何根据这三个组件进行算法实现。

# 实验与分析

## 实验环境

本文采用了斯坦福大学提供的Live Journal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 |

为了快速了解数据集的特征，我们首先在全集数据（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 |

## 实验结果

### 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。

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，我们的正确率计算公式为：

其中。即我们将比对每个顶点实际计算的值跟真实值是否相同，计算正确的节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，我们的正确率计算公式为：

其中为本系统中的TC算法计算出来的值，为图真实的TC值。

有了这些定义后，这三个算法的准确率在10组数据集上的表现如图所示：

### 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，我们分别测试了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。

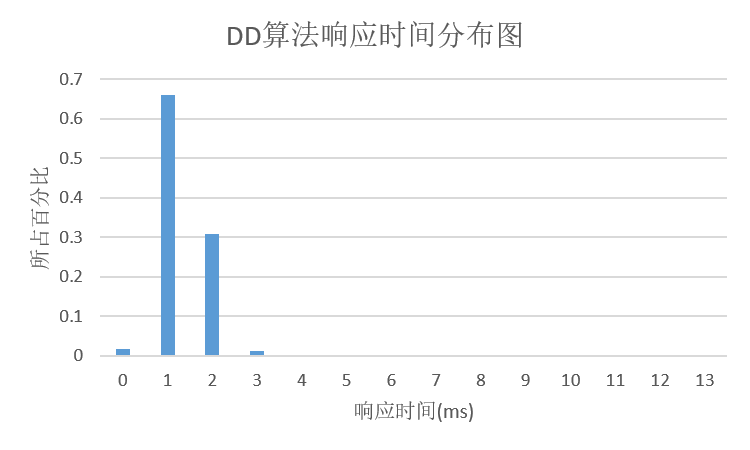
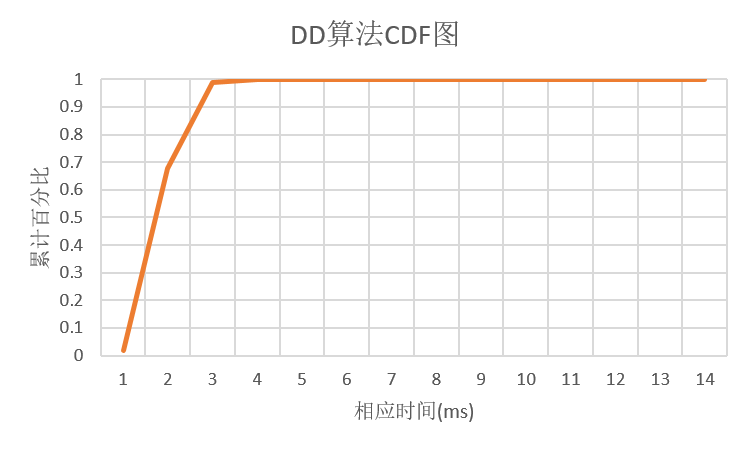


图 6‑2 DD算法实时性图

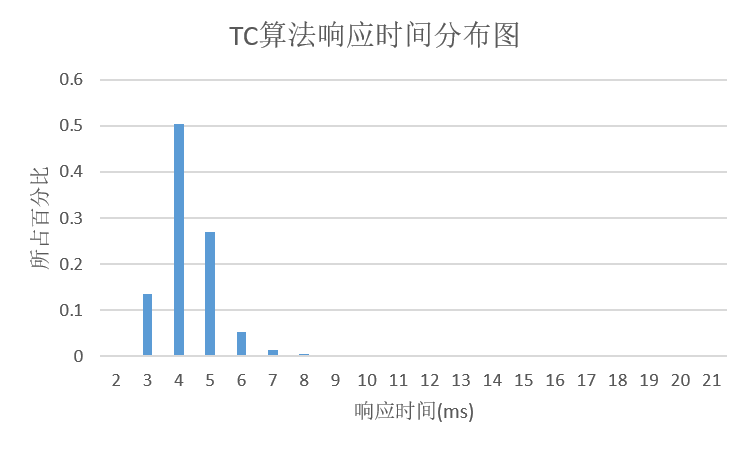
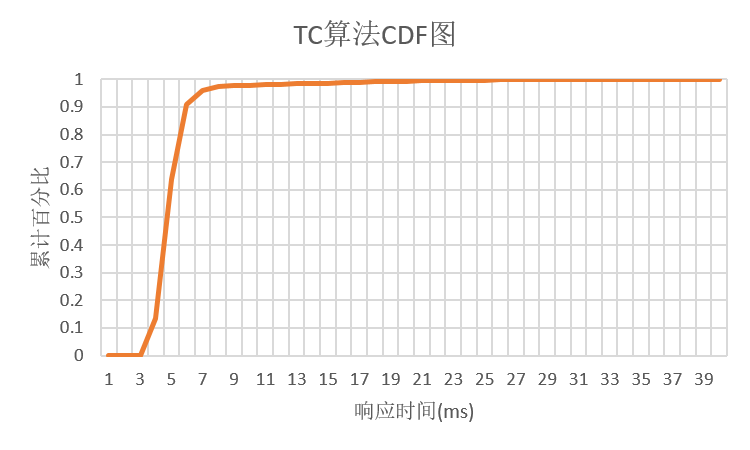


图 6‑3 TC算法实时性图

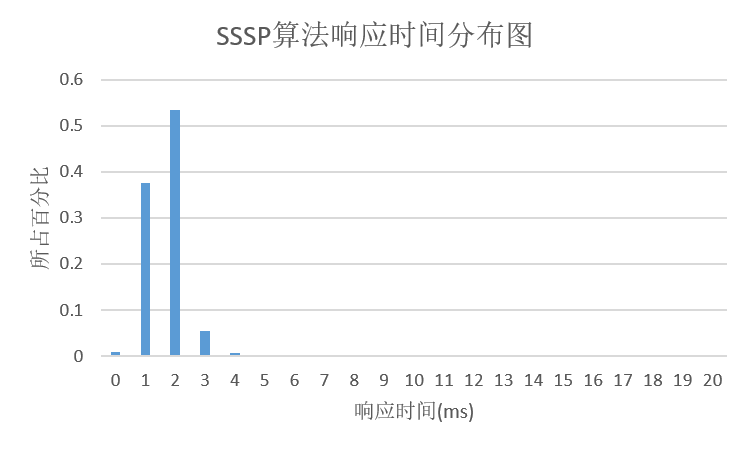
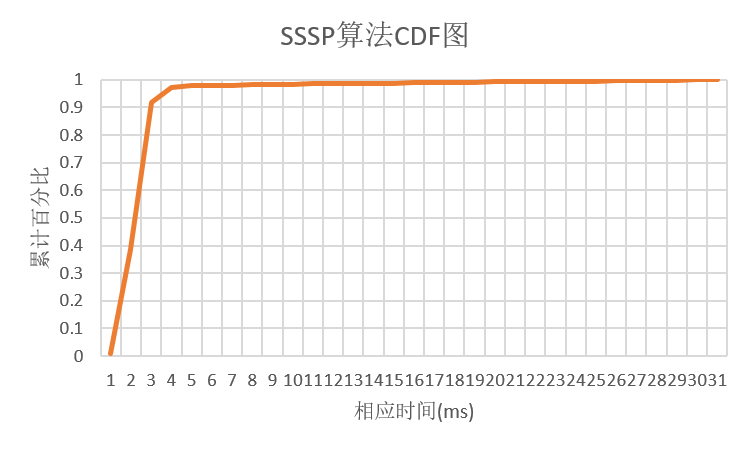


图 6‑4 SSSP算法实时性图

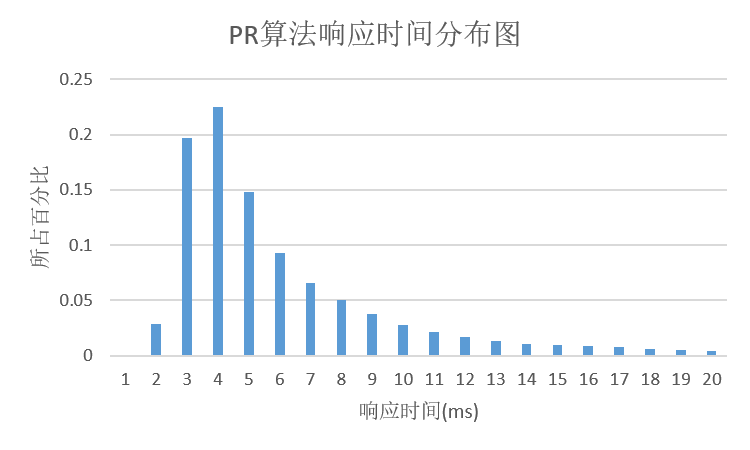
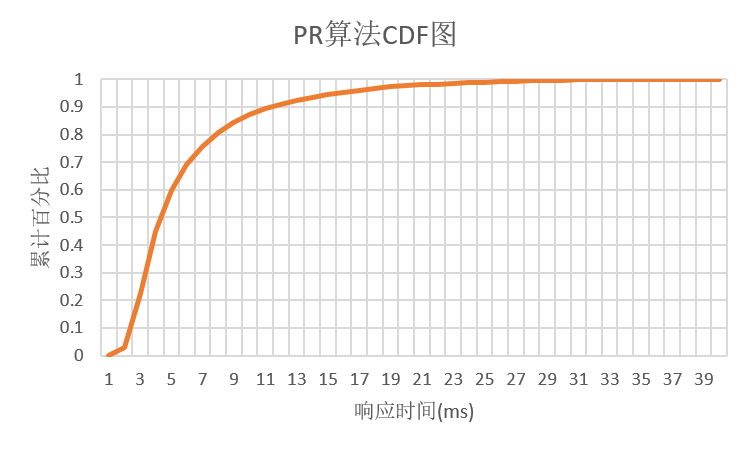


图 6‑5 PR算法实时性图

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他的顶点；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。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，我们还发现尽管不同算法的响应时间不同，但其分布符合长尾效应。这是因为自然状态下的图数据分布是符合power-low规则的，而符合power-low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，他们的更新代价很高，而大部分节点的邻接点都很少，所以更新的很快，因此大多数节点的响应时间都很短，只有极少数节点的响应时间很长。

### 更新冲突概率

在3.3节关联状态的存储和更新时，我们提出采用基于细粒度锁的并行更新策略，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，会出现更新冲突现象。为了评估这种更新冲突的概率，我们设计了如下实验，实验架构图见图6-5所示。本实验中我们首先将全集数据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的提高计算节点之间发生更新冲突的概率。



图 6‑6 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，我们分别测试的DD、TC、SSSP和PR算法，他们的实验结果如图6-6所示。

## 本章小结

# 结束语

## 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、编程模型和计算模型三个方面总结了解决海量图数据所需的关键技术，同时按照图处理模式的不同，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计算模型和系统，并且总结了在面对海量动态图数据时，以BSP计算模型为代表的批处理图计算系统所面临的问题。
* 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的动态图计算模型，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95%以上，90%以上的请求都能够在12ms内返回，符合实时性要求。

## 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，分析了他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的流式图计算模型，然而此模型对于无法满足着三个特征的算法不再适用，因此模型还有很大的提升空间，可以再充分考虑其他图算法，采用归纳总结的方式抽象出他们的基本特征，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，而对于其他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的流式图计算模型以及编程接口，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.

# 发表文章目录

# 作者简历

# 致谢