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

研究生学位论文中期报告

报告题目 面向流式图计算系统的设计与实现

学生姓名 段世凯 学号 201428015029007

指导教师 王伟 职称 副研究员

学位类别 工学硕士

学科专业 计算机软件与理论

研究方向 网络分布计算理论与技术

培养单位 中国科学院软件研究所

填表日期 2016 年 12 月 25 日

中国科学院大学制

填 表 说 明

1. 本表内容须真实、完整、准确。
2. “学位类别”名称填写：哲学博士、教育学博士、理学博士、工学博士、农学博士、医学博士、管理学博士，哲学硕士、经济学硕士、法学硕士、教育学硕士、文学硕士、理学硕士、工学硕士、农学硕士、医学硕士、管理学硕士等。
3. “学科专业”名称填写： “二级学科”全称。

报告提纲

一、学位论文进展情况，存在的问题，已取得阶段性成果

二、下一步工作计划和内容，预计答辩时间

三、已取得科研成果列表（已发表、待发表学术论文、专利等）

面向流式图计算系统的设计与实现

目录

1 研究背景、意义和现状

1.1 研究背景与意义

图是计算机科学中常用的一类数据结构，它能够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 2014 年第一季度 Facebook 包含了 12.3 亿个活跃用户，每个用户平均好友 130 个；web 链接图顶点数达到 T 级，边的个数达到 P 级[15]。

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。

现在有很多成熟的图计算系统，例如 Google Pregel, Spark GraphX, 这些图计算模型都采用了分布式的集群和 BSP 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。[16]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。为此本文提出了**基于状态更新的动态图计算模型**，能够在原有状态的基础上，并发增量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，这种模型既能够实时的反应图中间计算结果，也采用并发更新的方式显著提高了计算效率。

1.2 图计算模型现状

1.2.1 批处理图模型

图数据能够丰富表达事物之间的关联情况，被广泛运用于社交分析、商品推荐、欺诈检测等各个领域。图算法一般需要多次迭代，计算过程也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 MapReduce 及其开源实现的 Hadoop 为代表的传统面向数据并行（Data-Parallel）的计算模型难以对图数据和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。计算模型如下图所示：

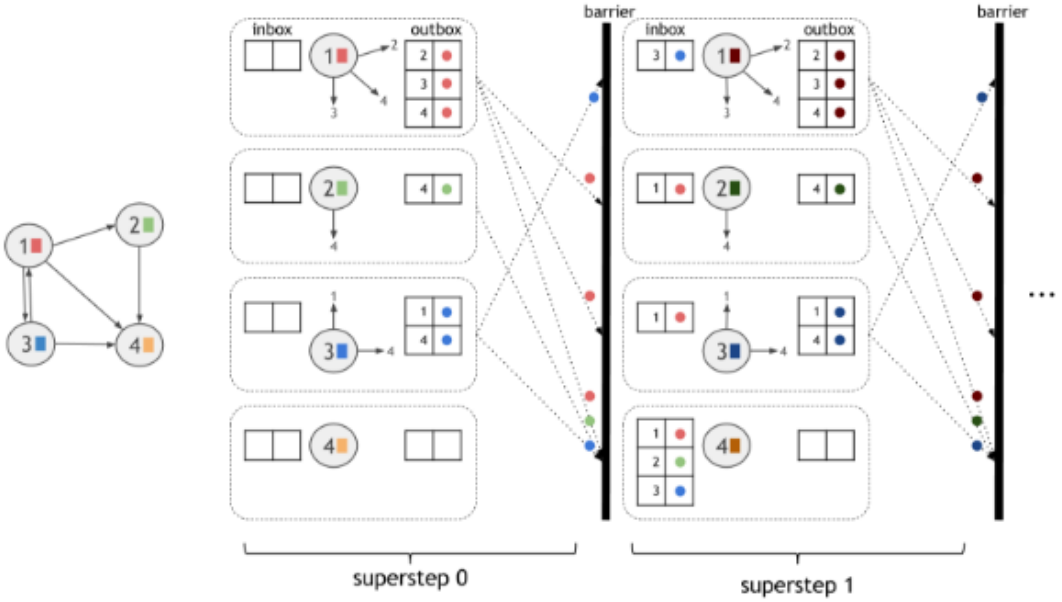

图 1-1 Pregel 计算模型

1.2.2 流处理图模型

流计算最初出现在 20 世纪末，是为了解决数据量大，数据不能完全保存在内存中的计算问题。通过将数据集转化为数据流，就可以依次处理流过的数据，而无需在内存中存储数据集，这样就解决了数据量大的问题。一开始，能够解决的问题领域也比较有限，主要都是针对数据流的计算。数据流上的算法研究成果有很多，如 computation of frequency moments [1]，histograms[2]，Wavelet transforms[3]，distinct elements count[4]等

由于图数据和图算法的复杂性，图流计算晚几年才出现。到目前为止，图流上进行的图算法有 Connectivity，Triangle Count[5]，PageRank[6]，Matching 等

无论是图流计算还是数据流计算，研究都集中在估计计算。估计计算主要有以下优点：a. 可以不存或者存储少量的数据，相对于整个图数据来说。这点对于 Internet 和社交网络等方面的分析有重大意义，因为它们的顶点和边数量庞大，甚至可能存在 2^{64} 条边，完全存储数据集是不现实的。b. 更新时间短，流算法对每个流数据要进行相应的计算和更新，而采用估计的流算法更新和计算时间短。在分析流计算估计模型之前，我们先看一下流模型。

依据流中数据的表达形式，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。例

如，流图中的每一项为(x, U)，如果 U 为＋，就将 x 加入 D，如果 U 为－，就将 x 从 D 删除。这种模型更符合现实的一些场景，近期的参考文献也将这种模型成为动态流(dynamic stream)。

现在我们转向流的估计计算模型，下列模型主要是针对图流进行分析，当然一些模型可以应用于数据流，这里不细区分。依据模型是否存储流中的数据，将模型分为两类：

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，见[5]，该论文定义了两类图流 arbitrary stream 和 incidence stream，并分别提出了 1-pass 和 3-pass 采样算法，以 arbitrary stream 的 1-pass 为例，该算法可以通过令采样次数 $s \geq \frac{3}{\varepsilon^2} \cdot \frac{|T_1| + 2 \cdot |T_2| + 3 \cdot |T_3|}{|T_3|} \cdot \ln\left(\frac{2}{\delta}\right)$，使得 $(1-\varepsilon) \cdot |T_3| < \tilde{T}_3 < (1+\varepsilon) \cdot |T|$ 的概率为 $1-\delta$，其中 $0<\varepsilon<1$，$0<\delta<1$，$\tilde{T}_3 = \left(\frac{1}{s}\sum_{i=1}^{s}\beta_i\right) \cdot |E| \cdot (|V|-2)$

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。依据做概要的方式，可以将该模型分为三类：

- 生成树(spanner) [8, 9]：针对图流来说，该模型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]：针对图流来说，该模型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) ：该模型又分为线性草图(linear sketch)[12, 13]和同构草图(homomorphic sketch)[14]，针对图流来说，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。

基于流的特征，我们可以将流图计算扩展到实时计算上，基本上上述的模型与算法均可以应用在实时计算。这样，就可以满足现代社会日益增长的实时计算的需求，研究有意义。

1.3本文的研究贡献

由 1.2 的两小节可知，针对静态图批处理模型和动态图流处理模型，现有的工作已经很多。对于静态图批处理模型，它应用的场景一般是假设图数据是稳定的，适合离线的图数据分析，而对于图数据是动态变化的场景则不适用；而针对动态图的流处理模型，现有论文的研究工作主要集中在如何采用抽样和化简的方式来进行估算。虽然已有大量文献通过各种优化的方式提高了这种估算的精度，但这种估算毕竟不是准确计算，在对结果要求苛刻的场景中缺乏可信度。

针对上述问题，本文工作主要有以下几点：

（1）本文分析了现有的图计算模型:**全局静态的批量处理模型**和**流式的增量处理模型**，分析

总结了现有的批处理模型的特点和应用场景，同时给出了面向流式图数据的处理模型的定义、解决方案和应用场景，最后对比分析两者之间的区别和联系；
（2）针对现有的图计算模型的优缺点，本文设计了一套针对流式图数据的**基于状态更新的动态图计算模型**，很好的解决了海量动态图数据的处理和计算问题，这使得用户不需要将全部的图数据导入内存之后再进行计算，能够在有限的内存内完成对无限的图数据流的处理，而且这种计算是实时的，即能够在计算过程中实时返回计算结果，而无需等到全部计算结束时才返回计算结果；
（3）本文在自己建立的流式图处理的框架上面，实现了几个典型的**图算法**，并且结合具体场景讲述了这些算法的用途，最后给出了这些算法的评测方案和评测结果。

2 学位论文进展情况（系统设计与实现）

2.1 系统设计

2.1.1 框架设计

传统的面向批处理的图计算框架，是在全部静态图数据上进行计算的。而我们提出的针对流式的图计算框架，是希望能够处理动态图数据，这使得原有的解决方案无法满足现有的数据类型。为此我们需要构建新的图计算框架。本文设计的面向流式图数据的计算模型如下图所示：

<table>
<tr><td>应用层</td><td colspan="6">Application</td></tr>
<tr><td>服务层</td><td colspan="6">Library</td></tr>
<tr><td>API层</td><td colspan="6">Graph Streaming</td></tr>
<tr><td rowspan="2">核心层</td><td colspan="3">Graph</td><td colspan="3">Streaming</td></tr>
<tr><td>Storage Model</td><td>Partition Model</td><td>Compute Model</td><td>Compute Model</td><td>Partition Model</td><td>Window Model</td></tr>
<tr><td>引擎层</td><td colspan="6">Specific Engine</td></tr>
</table>

图 2-1 面向流式图数据的计算框架

框架的核心组件主要有：

- **Application**：面向用户的上层运用，这些运用涵盖了典型的使用场景，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；
- **Library**：框架提供给用户使用的丰富的库函数和图算法。诸如 PageRank, Triangle Count, Connected Components 等算法包都会在该层中体现；
- **Graph Streaming Model**：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模型，该层需要充分考虑图计算和流处理的特点，针对图计算的核心问题：存储、切分和计算，以及流处理的核心问题：计算、分片和窗口，能够很好的将两者融

合起来，为上层用户提供一个统一的视角，**构建面向流式图数据的计算模型**；

- **Specific Engine**: 最底层的具体的引擎，例如可以借用现有的 Spark 或 Flink 这样的分布式并行计算框架作为整个系统的底层执行引擎。

本文的工作重心是希望能够很好的融合流处理和图计算这两个问题，希望能够提供一个面向流式图数据的计算模型，并且在该模型之上构建丰富的库函数，方便用户能够根据具体的应用场景来选择合适的图算法，最后会以欺诈检测这样一个具体的应用来演示和验证本文系统的正确性。

2.1.2 融合模型

传统的图计算模型中，图数据是静态的，即在计算的过程中不会发生变化，此时需要图计算和批处理进行融合，如在 1.2 节中分析的同步计算模型，异步计算模型和单机模型都是图计算和批处理的融合，而本文的融合模型是**图计算和流处理的融合**。本文提出**基于状态更新的动态图计算模型**，将动态图在每个时刻抽象成一个状态（***State***），事件（***Event***）触发了图由一个状态转变为另外一个状态。如下图所示：

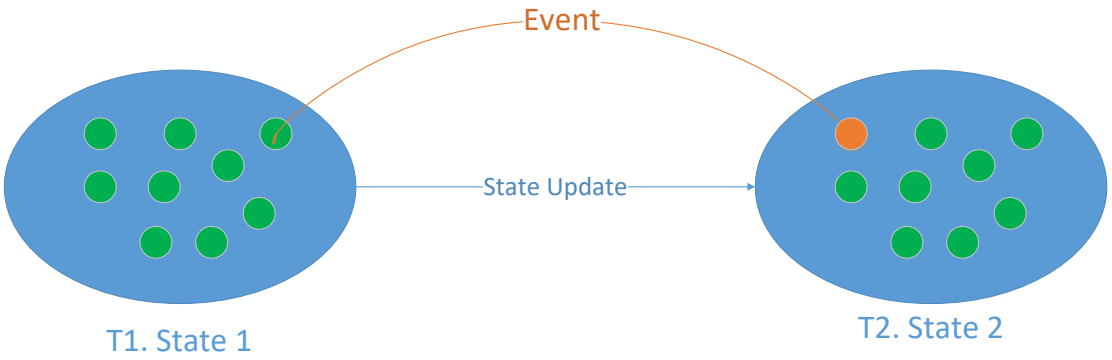

图 2-2 状态更新图

基于状态更新的流图计算模型有如下几个定义：（1）状态（***State***）：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，也可以使用用户自定义的特征信息来体现，状态反应了用户的关注点；（2）事件（***Event***）：触发图由 T1 时刻的***State***1 转换为 T2 时刻的***State***2 的事件，例如在 T2 时刻新增加了一条边，将使得图由***State***1 经过某种运算得到***State***2。（3）更新（***Update***）：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***State***1 转换成***State***2。

传统的批处理计算模型（如 BSP 模型）如下图所示，将计算过程定义为一系列的超步，在每个超步内，各个顶点之间进行通信和计算，并且更新各自的状态并进入下一个超步。相比较本文提出的基于状态更新的流图计算模型，BSP 模型是通过在静态稳定的图数据上不断的进行迭代直至达到收敛条件，它无法感知诸如新增边或顶点这样的外来事件。而且在每一步的计算中，所有活跃的顶点都会参与计算，通信和计算代价较高，而我们的基于状态更新的流图计算模型只针对新增的事件进行状态更新，代价较小。

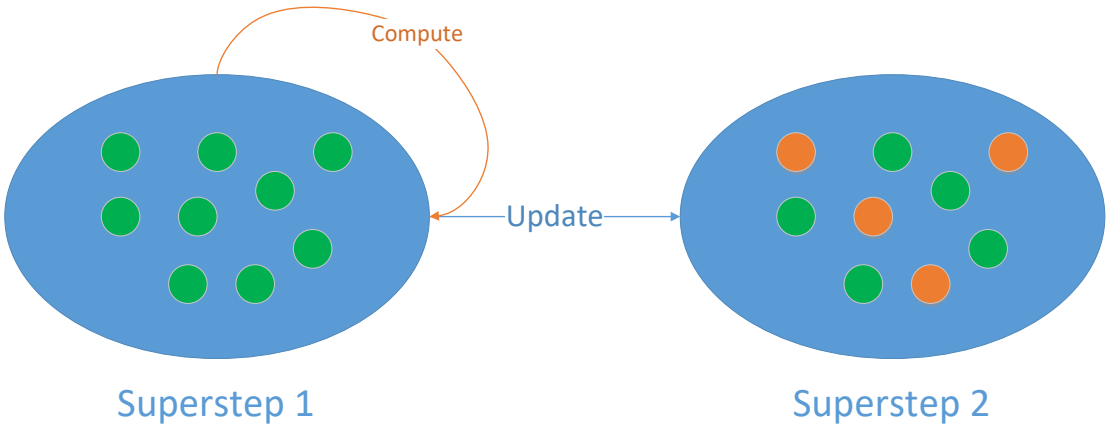

图 2-3 BSP 模型图

下面以统计图中各个节点的度为例，阐述如何使用基于状态更新的模型解决流式图数据的计算问题。假设有图$G=(V,E)$，其中$V=\{v_1,v_2,\dots,v_n\}$是顶点集合，边$E=\{e_1,e_2,\dots,e_m\}$是边集合，边按照$e_1,e_2,\dots,e_m$的顺序到达，现在希望在该动态图上统计各个节点的度。具体方法如下：

（1）定义图的$\boldsymbol{State}=\{s_1,s_2,\dots,s_n\}$，其中$s_k$表示节点$v_k$的度为$d_k$，即$s_k=(v_k,d_k)$；

（2）定义图的$\boldsymbol{Event}=\{z_1,z_2,\dots,z_m\}$，其中$z_k$表示新增边$e_k$，即$z_k=(e_k,add)$；

（3）定义图的$\boldsymbol{Update}$方法：

针对事件z_k，它表示新增了边e_k，设边e_k的源点和目标点分别为v_{k_1},v_{k_2}，其中$k_1,k_2\in 1,2,\dots n$。顶点v_{k_1},v_{k_2}在$\boldsymbol{State}$中对应的状态分别是$s_{v_{k_1}},s_{v_{k_2}}$。更新规则如下，首先检测$\boldsymbol{State}$中是否有这两个顶点的状态，如果没有，添加这两个顶点的初始状态，即记录当前这两个顶点的度为 1，如果有，则将这两个顶点的度分别加 1。算法伪代码如下：

Algorithm01 – Update Function

```
for z_k in Event
        e_k = (v_{k_1}, v_{k_2})
    if s_{v_{k_1}} in State
            s_{v_{k_1}} = (v_{k_1}, d_{v_{k_1}} + 1)
        else
            s_{v_{k_1}} = (v_{k_1}, 1)
        add s_{v_{k_1}} to State
    if s_{v_{k_2}} in State
            s_{v_{k_2}} = (v_{k_2}, d_{v_{k_2}} + 1)
        else
            s_{v_{k_2}} = (v_{k_2}, 1)
        add s_{v_{k_2}} to State
```

定义完三个基本组件之后，算 CW 法的整体框架已经构建起来，后续具体的执行过程如下：图中的边$e_k=(v_{k_1},v_{k_2})$以流的形式进入系统。算法接收这些边流信息，并且根据自定义的更新函数（$\boldsymbol{Update}$）不断的更新自定义的状态值（$\boldsymbol{State}$）。在更新过程中，用户可以在任意时刻查询图的状态，不需要等到全部执行结束时才能向用户反映结果—这种查询是

即时的。而且在更新时，不需要在整个图上进行更新。

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。状态可以存储在本地节点中，也可以分布式的存储在多机节点上。

定义完状态之后，接下来要解决的问题是如何存储状态和更新状态。因为状态是用户自定义的特征信息，所以该状态一般是直接放在内存中存储。可以按照状态的组织形式，分布式的存储在多个节点上，因此在更新时，也可以并发的对状态进行更新，然后再收集各个节点状态，向用户反馈状态的持续更新过程。注意到这种状态的存储可以是并发的存储和更新的，所以充分利用了分布式计算框架的特点，提高计算效率。状态的并发更新过程如下图所示：

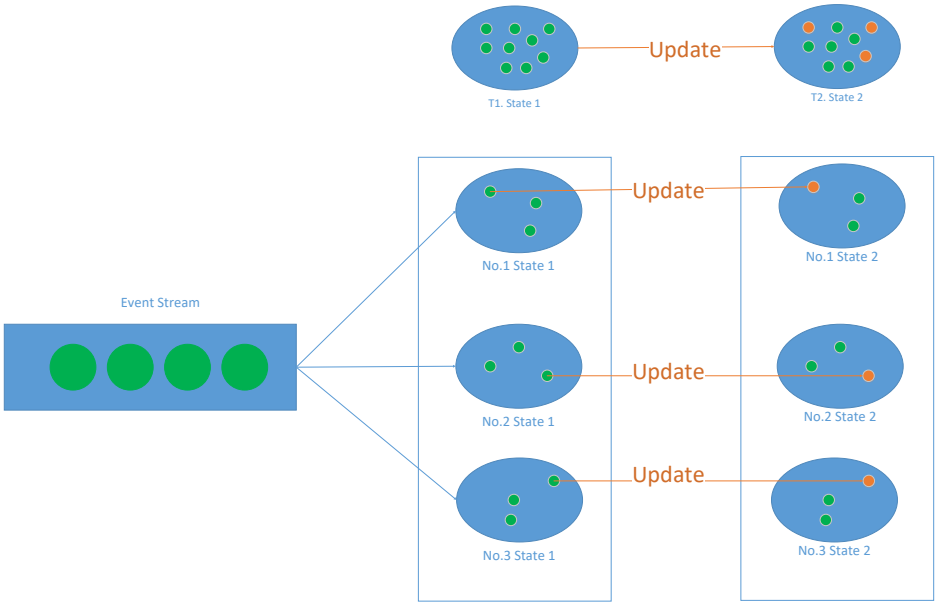

图 2-4 状态分布式存储更新图

该图演示了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），分解到不同的计算节点上（如图所示的 No.1, No.2, No.3 三个计算节点），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的状态从***State***1 转换到了***State***2），这样使得图由 T1 时刻***State*** 1，转换成了 T2 时刻的***State*** 2。

2.1.3 算法设计

2.1.4 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。

在社交网络分析中，社交网络中的三角形数目越多，表明社区内人物之间的联系越紧密。如常见的微博粉丝网络如下左图所示，而微信朋友圈网络如下右图所示，图中橘红色标出的三角形即为 Triangle Count 算法中需要统计的三角形。

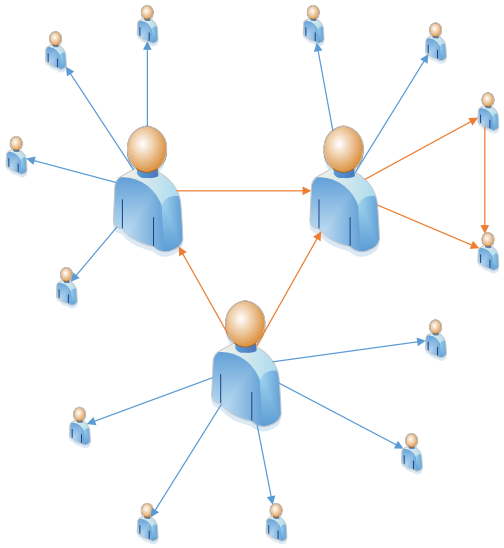

图 2-5 微信朋友圈网络

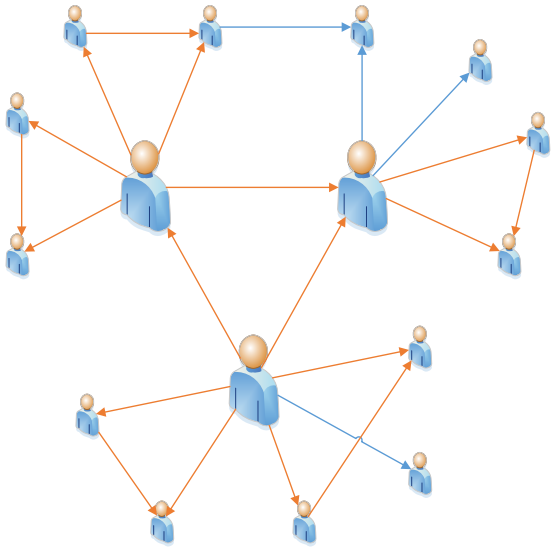

图 2-6 微博粉丝网络

传统的 Triangle Count 算法是在**静态**的数据集上，经过多次的迭代统计出来。一个 Naïve 的算法如下表所示：

Algorithm-02 Naïve Triangle Count

$count = 0, triangle = []$

$for\ v_a\ in\ V$

  $for\ v_b\ in\ Neighbor(v_a)$

    $if\ v_b > v_a$

      $for\ v_c\ in\ Neighbor(v_b)$

        $if\ v_c > v_b\ and\ v_c\ in\ Neighbor(v_a)$

          $count = count + 1$

          $triangle = triangle + (v_a, v_b, v_c)$

基于朴素的 Triangle Count 算法，部分文献都做了改进，显著降低了算法的时间和空间复杂度。但是此类算法是在原来大批量的静态图数据上进行的迭代运算，需要多次随机访问节点信息，而且图是静态的，无法感知新增边或节点的情况，再之如果图数据量过大时，每轮的迭代需要再超大规模的图上进行，效率较低。下面介绍如何在 2.3 节设计的基于状态更新的流图计算模型上实现 Triangle Count 算法。

由前文所知，基于状态更新的流图计算模型，有$\boldsymbol{State}$, $\boldsymbol{Event}$, $\boldsymbol{Update}$三个重要的概念。定义好这三个组件之后，就能够实现特定的算法。针对 Triangle Count 算法，这三个组件的定义如下：

（1）$\boldsymbol{State}$:图的$\boldsymbol{State}$由每个顶点对应的邻接点的信息组成，即$\boldsymbol{State} = \{s_1, s_2, \dots, s_n\}$，其中$s_k$表示节点$v_k$的邻接点为$N_k$，节点$v_k$构成的三角形的数目为$t_k$，$s_k = (v_k, N_k, t_k)$;

（2）$\boldsymbol{Event}$:图的$\boldsymbol{Event}$为图中新增了一条边，即$\boldsymbol{Event} = \{z_1, z_2, \dots, z_m\}$，其中$z_k$表示新增边$e_k$，$z_k = (e_k, add)$;

（3）$Update$:图在动态变化过程中，$\boldsymbol{State}$的更新过程如下：

Algorithm-03 Dynamic Triangle Count

$for\ z_k\ in\ Event$

$N_1 = [], N_2 = []$

$e_k = (v_{v_{k_1}}, v_{v_{k_2}})$

if $s_{v_{k_1}}$ *in State* $N_1 = N_{v_{k_1}} + v_{v_{k_2}}$

else $N_1 = v_{v_{k_2}}$

if $s_{v_{k_2}}$ *in State* $s_{v_{k_2}} = N_{v_{k_2}} + v_{v_{k_1}}$

else $N_2 = v_{v_{k_1}}$

$cross = N_1 \cap N_2$

for c in cross

$s_c = (c, N_c, t_c + 1)$

$s_{v_{k_1}} = (v_{v_{k_1}}, N_1, t_{v_{k_1}} + |cross|)$

$s_{v_{k_2}} = (v_{v_{k_2}}, N_2, t_{v_{k_2}} + |cross|)$

2.1.5 Connected Components

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**。如下图所示中，a)有向图总共有三个连通分支：{1,2,4,5},{3},{6}。{1,2,4,5}中的顶点是相互可达的。而顶点 6 不能从顶点 3 到达，则顶点集{3,6}不能形成连通分支；b)无向图中总共有三个连通分支：{1,2,5}，{4}，{3,6},在每个连通分支内，任意两个节点都是可达的。

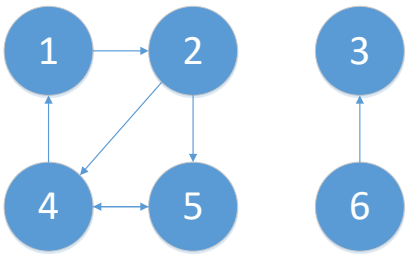

图 2-7 a)有向图

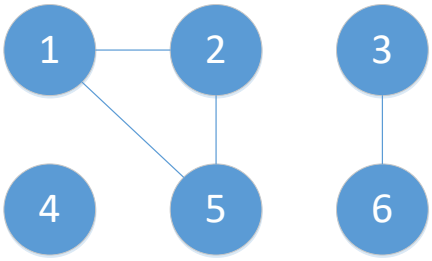

图 2-8 b)无向图

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使用，是图的基本问题。

关于求解连通分支的问题，现有的研究甚多。下面将介绍如何在流式的动态图数据上，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***, ***Update***，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，$\boldsymbol{State} = \{s_1, s_2, \ldots, s_n\}$，其中$s_k$表示第 k 个连通分支，$s_k = (v_{k_{min}}, \{v_{k_1}, v_{k_2}, \ldots, v_{k_r}\})$，其中$\{v_{k_1}, v_{k_2}, \ldots, v_{k_r}\}$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，即$\boldsymbol{Event} = \{z_1, z_2, \ldots, z_m\}$，其中$z_k$表示新增边$e_k$，$z_k = (e_k, add)$;

（3）***Update***:图在动态变化过程中，***State***的更新过程如下：

Algorithm-04 Dynamic Connected Components

$for\ z_k\ in\ Event$

$e_k = (v_{k_1}, v_{k_2})$

$if\ s(v_{k_1})\ in\ State\ and\ s(v_{k_2})\ in\ State$

$union(s(v_{k_1}), s(v_{k_2}))$

$if\ s(v_{k_1})\ in\ State\ and\ s(v_{k_2})not\ in\ State$

$union(s(v_{k_1}), v_{k_2})$

$else\ if\ s(v_{k_1})\ not\ in\ State\ and\ s(v_{k_2})\ in\ State$

$union(s(v_{k_2}), v_{k_1})$

$else$

$s_k = (min\{v_{v_{k_1}}, v_{v_{k_2}}\}, \{v_{v_{k_1}}, v_{v_{k_2}}\})$

$add\ s_k\ to\ State$

在上述算法中，$s(v_{k_1})$表示节点v_{k_1}所在的连通分支，$s(v_{k_2})$表示节点v_{k_2}所在的连通分支，由于新增了一条边$e_k = (v_{k_1}, v_{k_2})$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。即算法中的$union()$函数；也可能存在其中一个节点之前未出现过，即不在任何一个连通分支内，则将其加入另外一个存在连通分支的节点中；如果这个节点都没有对应的连通分支，则这两个节点将构成新的连通分支，加入到原来的$State$中。

2.2 系统实现

2.2.1 框架实现

自以 Hadoop 为代表的 MapReduce 模型推出之后，大数据生态圈迎来一个井喷时代，各种相关的分布式计算框架涌现。诸如 Spark/Flink 这样高可靠、高性能的分布式内存计算框架，一经推出，在学术界和工业界都备受关注。经过几年的不断完善，已经逐步发展为一站式解决各种大数据问题的通用的编程框架，这不仅降低了开发者的学习成本，还为整个大数据领域提供了一个统一的处理平台。如下图所示是截止到目前为止（2016 年 12 月 19 日），Spark 和 Flink 的技术栈，可以看出他们都提供了丰富的编程接口来解决特定领域的问题，本文所关心的 Graph 和 Streaming 都有包含在内，但遗憾的是无论是 Spark 还是 Flink，Graph 组件提供的算法是运行在静态稳定的图数据上，无法处理流式的动态图数据。

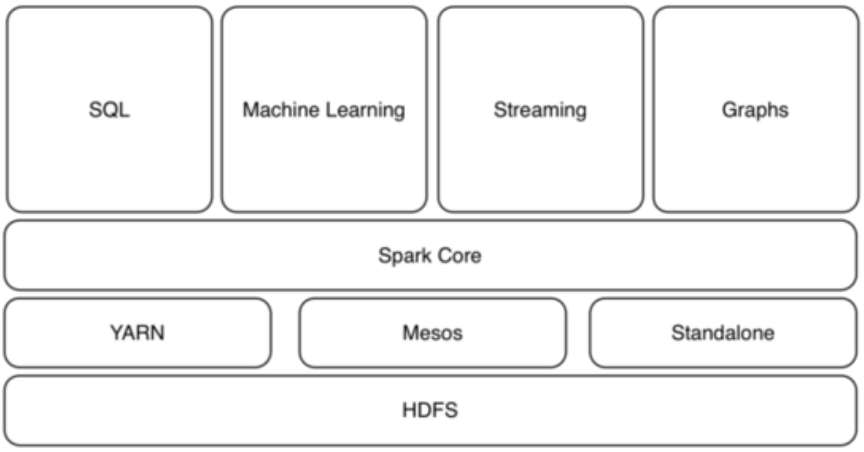

图 2-9 Spark 框架

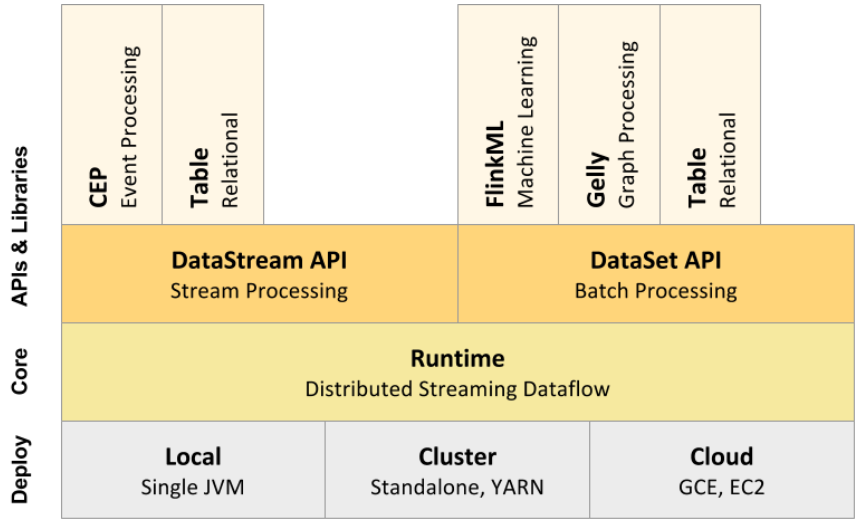

图 2-10 Flink 框架

整体来看，Flink 和 Spark 有许多相似之处，诸如都提供了 Streaming/ML/Graph 相关的 Library，都可以本地、集群或者在云上部署运行。但在某些方面也有所区别：（1）Flink 提供了精准的恰好一次的语义保障，这在失效恢复要求高的场景非常适用；（2）Flink 针对批处理和流处理分别提供了两种独立的 API（需要说明的是在运行时刻，批处理也被当成流处理来执行），而 Spark 中的流处理，本质上是微批次处理。（3）流处理中，Flink 提供了更多的窗口模型来应对更多的场景。基于以上 3 点，本文选择在 Flink 上进行实现。需要特别说明的是，本文的工作重心并不是要比较 Spark 和 Flink 的区别，而是希望能够在底层的执行引擎之上，构建一个相对统一的面向流式图数据的计算框架，搭建的这套框架也可以移植到 Spark 引擎之上。因此，本文所构建的整套系统的框架如下图所示：

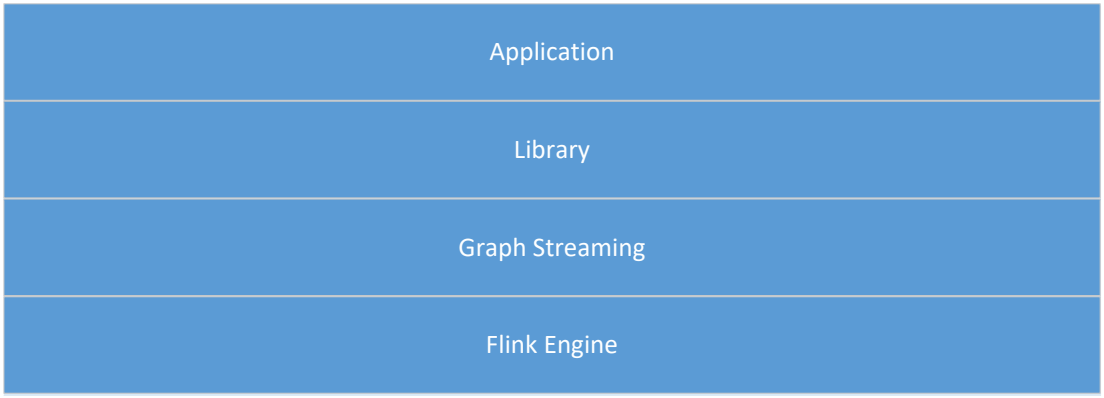

图 2-11 框架实现图

框架的实现步骤如下：

（1）**Flink Engine:** 深入学习 Flink Engine 的核心组件，尤其是 Graph 组件和 Streaming 组件，能够掌握 Flink 引擎 API 的使用规则，引擎的执行过程，并且能够修改引擎关键代码；

（2）**Graph Streaming**: 在 Flink Engine 上，实现 2.3 节定义的面向流式图数据基于状态更新的动态图计算模型，该模型对上层提供了一个统一的接口，用户使用该接口能够方便的查询、访问和修改流图数据，并且构建函数库，对下层封装了流处理和图处理的操作细节，充分利用底层的流处理 API，重写图处理的 API（因为原来的图处理是建立在批处理上，所以在本模型中不能够被使用），该层是系统的核心层；

（3）**Library:** 在 Graph Streaming 上，定义算法的基本框架和组件，方便用户能够轻松的实现自定义的算法，同时提供几个经典的图算法供用户直接使用；

（4）**Application**: 结合一个具体的场景-欺诈检测，来自上而下的搭建一套完整的解决方案，来验证系统的正确性和有效性。

2.2.2 模型实现

在 2.3 节融合模型中，我们详细介绍了基于状态更新的动态图计算模型。由前述所知，该模型的三个核心组件：***State***, ***Event***, ***Update***是实现模型的关键。下面首先将详细阐述这三个组件的定义和使用方法，然后以统计动态图中边的数目为例，讲解如何将这三个组件组合起来实现特定需求。

（1）***State***

State定义了图当前的状态，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，也可以使用用户自定义的特征信息来体现。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图数据，只需要存储精巧的状态信息即可反应图的特征信息。

例如我们希望统计图的边数，此时***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。

在这里，我们将状态抽象成一个接口，该接口只有一个方法，即返回当前时刻的状态信息，用户可以扩展该接口来实现更加复杂的状态信息，该接口图如下：

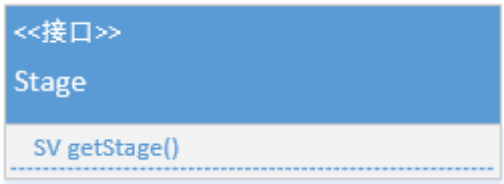

图 2-12 ***State***接口图

（2）***Event*** & ***Event*** Stream

Event代表一个能够影响图数据的事件，那么连续的图数据流就可以抽象成连续的事件流（***Event*** Stream）。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***）。这样总共可以组成 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。类关系图如下：

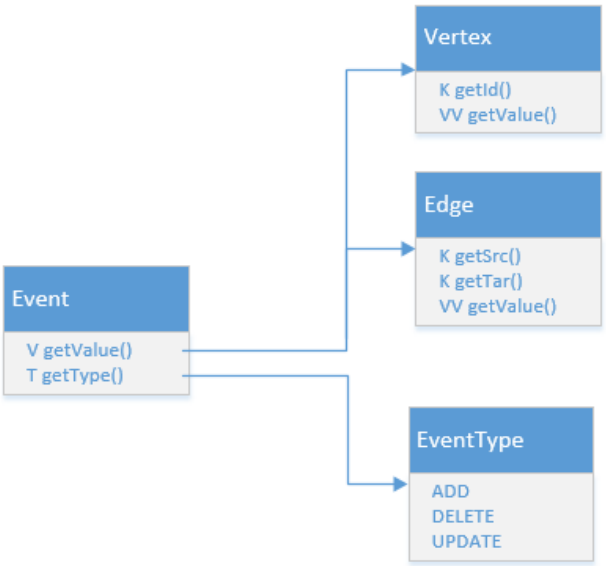

图 2-13 ***Event***类图

定义完基本事件之后，就可以由基本事件来形成事件流。事件流是同种类型的事件，按照特定顺序（如按照时间顺序或者到达系统的顺序）形成的流。系统只需按照一定规则处理该事件流就可以完成对动态图进行处理。

（3）*Update*

更新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。它可以称之为状态更新的图计算模型的驱动程序，驱动图从一系列的时间流转换成一系列对应的状态流。

例如我们希望统计图的边数，状态就设置为一个计数器反应当前时刻图的边数，而***Update***函数每接收一个事件，就根据该事件的类型（add, delete, ***Update***）对计数器进行修改。

更新函数接收一个状态和一个事件，然后将该事件应用在该状态上，返回一个新的状态，用户可以实现该接口以实现更复杂的转换过程，该接口图如下：

图 2-14 ***Update***接口图

下面我们以统计图的边数为例，讲解如何组合使用这三个组件。

（1）定义图的***State***，在***State***内部设置一个计数器（counter），该计数器反应当前时刻图中的边的数目；

（2）定义图的***Event***，该***Event***是连续变化（新增或删除）的边流；

（3）定义图的***Update***，该函数根据当前接收的***Event***类型来改变图的***State***：如果是“新增”边则将***State***中的 counter 增加 1，如果是“删除”边则将***State***中的 counter 减少 1。

通过前两步的定义和第三步的转换，就可以将原来的事件流映射成现在的状态流，该状态流能够及时反映当前图中接收到的边的数目信息。

2.3 系统验证

2.3.1 实验设计

（1）实验环境

利用现有的实验室机器，搭建 Flink 集群进行测试。实验环境如下：

表格 1 实验环境表

操作系统	**CentOS 6.4**
内存	16G
CPU	4 核 2.5GHz
网络	千兆网卡
硬盘	100G
机器数目	16 台
JDK	1.7

（2）实验目的

a. 验证基于状态更新的动态图计算模型的**正确性**。利用该模型实现的算法能够正确运行。

b. 验证本文实现的算法的**正确性**。算法能够处理流式的图数据，而且计算结果符合预期目标。

c. 验证系统的**稳定性**。在处理流式的海量图数据时，系统能够稳定运行，无内存泄露情况。

d. 验证系统的**实时性**。算法能够在执行过程中，实时反馈计算结果，延迟在毫秒级别。

e. 验证系统的**扩展性**。计算节点从 2 台扩展至 16 台，系统能够合理的分配资源并行计算。

（3）实验方法

采用对比实验，运行静态图数据的批处理模型算法，动态图数据的估算算法和本文的基于增量更新的动态图计算模型算法，对比三者算法在正确性、稳定性、实时性和扩展性上的差距，并且分析实验结果，给出实验结论。

3 总结、下一步计划和预答辩时间

3.1现有工作总结

从十月中旬接收这个项目为止，到现在 12 月中下旬，减去中间去成都出差两周时间，在这总共两个多月的时间里，我阅读了大量的论文，Flink 的技术文档，Flink 的源码和其他相关的开源代码，终于在 12 月份有了实质性的突破。

现阶段已经基本完成：

（1）本文第一部分：分析对比批处理流图模型和流处理流图模型的优缺点；

（2）本文第二部分：建立面向流式图数据的处理模型和框架；

（3）对于第三部分，在该模型上实现图算法，目前只实现了 Triangle Count 算法。

现阶段存在的问题：

（1）对于现在建立的模型过于简单，无法处理更为复杂的情况，需要在后期的工作中仔细斟酌模型的修改；

（2）对于评测方案设计的不是特别细致，后期需要完善完整的评测方案。

3.2接下来的工作计划

接下来的工作重心主要集中在：

（1）其他几个核心算法-PageRank, Connected Components, K-Cores 等的实现；

（2）实现完算法之后，对这些算法和整个系统进行系统的测试，并且根据测试结果进行总结和反思；

（3）搭建完整系统之后，在该系统之上，构建金融反欺诈应用，利用该应用来验证整个系统的正确性，可靠性，稳定性和实时性。

3.3预答辩时间

2017 年 5 月

4 已取得的科研成果

已发表专利一项：《一种基于混合存储的流式数据自适应持久化方法》

5 参考文献

[1] Noga Alon, Yossi Matias, and Mario Szegedy, The space complexity of approximating the frequency moments. J. Comput. Syst. Sci. 58 (1999), no. 1, 137–147.

[2] Sudipto Guha, Nick Koudas, and Kyuseok Shim, *Data-streams and histograms*, ACM Symposium on Theory of Computing, 2001, pp. 471 – 475.

[3] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss, *Surfing wavelets on streams: One-pass summaries for approximate aggregate queries.* Proc. of the 27th VLDB, 2001, pp. 79 – 88.

[4] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *6th Annual European Symposium (ESA'02)*, pages 1–10, 2002.

[5] Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. Counting triangles in data streams. In *25th Symposium on Principles of Database Systems (PODS'06)*, pages 253–262, 2006.

[6] Atish Das Sarma, Sreenivas Gollapudi, and Rina Panigrahy. Estimating pagerank on graph streams. *J. ACM*, 58(3):13, 2011.

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

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

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

[10] 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.

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

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

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

[14] 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

[15] 袁培森,舒欣,沙朝锋,徐焕良. 基于内存计算的大规模图数据管理研究[J]. 华东师范大学学报(自然科学版),2014,05:55-71.

[16]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.

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

[18] 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.