密级：　　保密期限：

北京郵電大學

硕士学位论文

题目：基于分布式计算的大规模网络分析系统的研究

学　　号：2015110859

姓　　名：赵炳

专　　业：计算机科学与技术

导　　师：张雷

学　　院：计算机学院

年　月　日

独创性（或创新性）声明

本人声明所呈交的论文是本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中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京邮电大学或其他教育机构的学位或证书而使用过的材料。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并表示了谢意。

申请学位论文与资料若有不实之处，本人承担一切相关责任。

本人签名：日期：_____

关于论文使用授权的说明

学位论文作者完全了解北京邮电大学有关保留和使用学位论文的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属北京邮电大学。学校有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许学位论文被查阅和借阅；学校可以公布学位论文的全部或部分内容，可以允许采用影印、缩印或其它复制手段保存、汇编学位论文。（保密的学位论文在解密后遵守此规定）

保密论文注释：本学位论文属于保密在年解密后适用本授权书。

非保密论文注释：本学位论文不属于保密范围，适用本授权书。

本人签名：日期：_____

导师签名：日期：_____

基于分布式图计算的大规模网络分析系统

摘 要

随着互联网的飞速发展，原本相互独立或者关系稀少的数据联系越来越紧密，产生了大量带有复杂关系的数据。这些复杂的关系需要图分析技术才能准确的反映数据的真实状态。但是一方面现有的图计算技术都是对静态图的分析，然而现实世界中的数据模型都是时刻变化的，现有的静态计算模型不足以反映网络的真实状态。另一方面由于图的结构复杂，导致图计算的复杂度很难降低下来，分布式的通信成本也居高不下。这些原因一直制约着图计算的发展甚至是图分析方向的前景。

本文首先研究了动态图存储结构，提出了适用于时序图存储的分布式图存储架构 TSGraph，底层存储模型使用 BigTable 模型，并使用 HBase 作为后端存储。通过存储结构的优化，RowKey 的设计，查询索引的优化，该存储架构可以在时间维度上快速查询到动态图的某个静态快照，从而解决了动态图的持续性分析问题。

其次本文研究了增量式图算法和全量校正的方案，设计并实现了基于 Spark • GraphX 技术的增量图算法。实验表明该算法在一定的准确度丢失的容忍程度下，可以极大地提高图计算的执行效率。同时针对增量式算法会丢失准确度的这个问题，设计了全量校正的方案，可以保证计算结果一直在某个精度损失的容忍范围内。

最后基于论文提到的动态图存储系统和增量式计算系统，设计并实现了一个支持动态图计算的综合图分析平台 OpenGraph。该图分析平台主要包括数据的管理加载，图关系的查询，增量式算法的实时运行和图属性指标分析四个模块，该分析平台可实现大规模动态图的存储和分析。

关键词：TSGraph OpenGraph 动态图 增量计算 图快照 Spark

Research on Large Scale Network Analysis System Based on Distributed Graph Calculation

ABSTRACT

With the rapid development of the Internet, data that are originally independent or sparsely connected are getting closer and closer together, generating a large amount of data with complex relationships. These complex relationships require graph analysis techniques to accurately reflect the true state of the data. However, on the one hand, the existing graph calculation techniques all analyze the static graph. However, the data models in the real world change from moment to moment, and the existing static calculation models are not enough to reflect the real state of the network. On the other hand, due to the complicated structure of the graph, it is difficult to reduce the computational complexity of the graph, and the distributed communication cost is also high. These reasons have always restricted the development of graph computation and even the prospect of graph analysis.

This paper first studies the dynamic storage structure, proposes a distributed graph storage architecture TSGraph for temporal graph storage, the underlying storage model uses the BigTable model, and uses HBase as the backend storage. Through the optimization of storage

structure, the design of RowKey and the query index optimization, the storage architecture can quickly query a static snapshot of the dynamic graph in the time dimension to solve the problem of persistence analysis of the dynamic graph.

Secondly, this paper studies the incremental graph algorithm and the total amount of correction scheme, and designs and implements the incremental graph algorithm based on Spark GraphX technology. Experiments show that this algorithm can greatly improve the efficiency of graph computation under the tolerance of loss of accuracy. In the meantime, aiming at the problem that the incremental algorithm will lose the accuracy, a scheme of full correction is designed to ensure that the calculation result always falls within the tolerance range of a certain accuracy loss.

Finally, based on the dynamic storage system and incremental computing system mentioned in the paper, an integrated graph analysis platform OpenGraph is designed and implemented. The graph analysis platform includes four modules: data management loading, graph relation inquiry, incremental algorithm real-time operation and graph attribute index analysis. The analysis platform can store and analyze large-scale dynamic graphs.

KEY WORDS: TSGraph OpenGraph Dynamic Graph Incremental Calculate Graph Snapshot Spark

目 录

绪论

1.1 研究的背景与意义

第一章

随着信息技术的发展，互联网上的信息规模出现了爆炸式增长，如今仅仅在中国互联网网页已经达到 2360 亿[1]，而且用户数目也突破了 7 亿[1]。与此同时，互联网上的服务模式也越来越丰富，信息之间的联系越来越紧密。超连接图（hyperlink）[2]将不同的网页连接起来，知识图谱（knowledge graph）[3]将不同的实体（entity）联系起来，而社交网络（social network）[4]又将不同的用户连接起来。

这些大规模、高度结构化的数据很大程度上应了真实世界中的关系，具有很重要的研究价值。因此，相关领域也出现了大量的图分析算法，通过计算数据的结构化特征，可以提取出重要的信息，从而得到巨大的应用价值。其中，具有代表性的技术包括排序（ranking）[5]技术， 社区群体（community）分析[6]技术，话题（topic）分析[7]技术等。

在社交网络分析[8]、互联网搜索[9]以及知识图谱等领域里面。图由于其强大的高维度表示能力被广泛的使用。近些年来人们越来越意识到了对于图结构的存储和计算的重要性，一批图数据库和图计算框架相继涌现出来，并且还取得了不错的反响。这其中比较有代表性的有用于存储的 Neo4j[10]，TitanDB[11]和用于计算的 GraphLab[12]，Spark GraphX[13,14]等。

现有的分布式图计算系统通常是采用边分割或者点分割的方式将一个大图分布式存储到集群里单独计算机的内存上，被分隔开的点或者边都会在其他机器上存在备份。每个节点计算完毕后会把计算结果同步到其他有该节点备份的主机上面。文献[12]提出了一个基于图像处理的开源图计算框架 GraphLab，该框架是一个面向机器学习的流处理并行计算框架。GraphLab 将数据抽象成 Graph 结构，并且将计算过程抽象成 Gather、Apply、Setter 三个阶段，Gatther 阶段所有节点从邻接节点接收数据，并且进行计算，Apply 阶段将 Gather 节点计算得到的值发送到 Master 上，由 Master 汇总再进行计算并且更新节点值，Setter 阶段将会更新产生变化的节点相邻的边信息，并且通知相邻接点该节点已产生变化。文献[13]基于 Spark 的底层结构实现了一个分布式图计算框架 GraphX，它充分的利用了 Spark 的分布式内存模型 RDD 的优点，通过 Spark 的并行处理能力和 Pregel[15]的原理，可以实现大部分图计算任务，速度较 MapReduce 有了很大的提升。文

献[16,17]从图分割的角度出发，通过良好的划分算法可以有效的减少分布式系统的通信代价。

现在虽然已经有了增量图计算的方式，但是其算法适用范围受限，并且没有完整的平台系统支持图的存储和分析。因此研究出一套完整的支持大规模动态图存储和分析的系统就非常重要了。

1.2 国内外研究现状

1.2.1 图存储

目前基于图的存储主要有三个主流的方向，第一个是基于关系型数据库构建图数据库，这种类型主要是基于当前的关系型数据库做一些表设计上的修改，仍然使用扩展度非常高的 PostgreSQL 作为与之对应的查询语言。因为关系型数据库有着几十年的工程积累，通过大量的实践积累了相当丰富的设计经验，所以基于这方面的扩展还是有很多支撑的，近些年也一直有论文讨论相关的话题。然而，图查询的本质难题是数据高度关联带来的大量的随机访问，传统的关系型数据库很难解决这个问题，因为他们是面向磁盘优化，最大化的利用了磁盘顺序读写的优势，对于图数据的查询却未必适用。这一类数据库的一个典型代表就是微软的 GraphView[18]，

第二个是以 Neo4j 为代表的称之为原生的图数据库，这种数据库摒弃了关系型数据库的设计模式。主要特点是查询一个点的边或者是边上的端点时不用再次线性查找或者重新走一遍 B+树索引，而是直接用指针指向下一度的物理地址，基于关系的查找使用遍历链表的方式实现。它的双向链表结构在内存足够大或者是有 SSD 盘辅助的情况下查找性能还是不错的，但是在内存不够的时候，性能上会有稍微的折损，但是总的来说还是一个不错的选择。

第三个是以 TitanDB，JanusGraph 为代表的使用了 NoSQL 存储的分布式图数据库。目前的产品实现方案主要是在 NoSQL 数据库（比如 HBase，Cassandra）上封装了一层逻辑的图结构，存储和查询分离，目前的性能提升空间还有很大。

三中存储方式各有千秋，也各自有难以克服的缺点。

1.2.2 图计算

通用的大规模数据存储分析平台，大部分已经集合了并发执行，作业调度，容错管理等一系列复杂的功能，想要使用的话只要部署了相应的环境，并且调用通用的接口就可以实现大部分的算法的执行。目前比较知名的大规模分布式计算平台主要有 Google 提出的的 MapReduce[19]系统、微软的 Cosmos[20]、Apache 的 Hadoop[21]和 Spark[22]。这些通用的计算平台当初的设计目的是为了解决大量共有的海量数据分析过程中遇到的难题，因此设计目标比较宽泛，可以支持多种多样

任务的计算。如今，这些有名的计算系统已经被广泛的使用在了工业界和学术界，为信息化产业的进步贡献了很大的力量。

然而通用计算平台的设计目标宽泛，主要是为了解决业界遇到的大部分通用的海量计算任务，并没有明确的区分不同的计算任务之间的差别，导致这些通用的计算平台在某些特定的计算任务上表现的并不是很好，特别是在图计算领域，这个问题尤为突出。

1.2.3 动态图处理系统

动态图是具有时间属性的图，静态图相当于是动态图在某一具体时刻上的映射。相较于静态图来说，动态图更能显示现实世界中的种种联系，比如说互联网中的网页，每时每刻都有新的网页链接加入进来。人与人之间的社交关系，也是时时刻刻都产生着新的互动。因此想要更深入准确的显示现实世界的关系，可以及时处理动态图的图计算系统就显得尤为必要了，然而现有的研究往往还在静态图计算系统的研究上，主要是动态图计算难度大，涉及方面广，很难做出一个适用广泛的系统。

近些年以来，时序数据库的出现就是一个重要的标志，人们已经越来越意识到单凭关系型数据库已经很难在这个数据爆炸的时代分析各种大数据任务了。比如自动驾驶环境中汽车遇到的各种场景的收集分析，自动化交易算法持续不变的收集着交易场所的数据，智能空调实时的调整房间的温度。从 FaceBook 推出了 Beringei[23]以来，其他的工业界巨头也都纷纷的推出了自己的时序数据库产品，包括百度天工的时序数据库 TSDB，阿里的 HiTSDB 等。然而在图数据库方面，仍然没有一个时序图数据库的出现。虽然各大研究机构也都纷纷进行了各自的研究，但是从目前的产品来看，效果并不那么显著。其中比较著名的便是微软亚洲研究院的 KineoGraph[24, 25]，KineoGraph 从存储到计算，提出了一个方便的动态图计算框架，遗憾的是至今并没有听说在工业界使用。其他的一些关于动态图的计算还有 SpecGraph[26]、InscGraph，不论这些产品是否成功，但是在图计算系统的研究上都贡献了宝贵的经验。针对这些存在的问题，本文将研究适合动态图的存储结构和适合动态计算的图算法，从而填补目前在动态图计算方面的空白。

1.3 本文的研究内容

本文首先研究了图存储的底层结构和时序数据库的实现方法，提出了邻接表的 Key-Value 存储模式，结合存储结构的优化和索引技术的加持，设计了一个高效的、可用的可以存储动态图的存储架构。动态存储的基础是可以存储时序信息的 HBase 数据库，HBase 使用 BigTable 存储模型，采用 Key-Value 结构，在图查询方面有着得天独厚的优势。在深层次关系图查询方面采用 Key-Value 和广度优

先搜索与深度优先搜索结合的方式，极大的提高了深层次关系查询的效率。针对动态图的时间属性，可以使用 RowKey 优化和索引技术，这样就可以方便快速的得到整个时序图中的某个时间区段快照。

其次设计了主体存储与计算副本并存的体系结构。对于主体存储来说，最重要的是实现快速、精准的查询。对于快速查询的需求，主要是从图的底层存储结构，硬盘的加载以及索引技术加持，实现低延迟的查询。对于准确性的要求就需要数据库操作满足原子性、持久性、一致性和容错性等特点，也就是要支持事务，可以实现事务的提交与回滚。对于计算副本来说，最重要的就是要适合多次的迭代计算，可以保持数据与主体存储的一致。在这里主要采用 HDFS 与 Spark 的 RDD 存储结合的方式，每次迭代计算 RDD 都会检测和上次计算的差别，从而决定是否要进行下一轮计算。主体存储和计算副本之间采用消息队列系统通信，既可以保证数据的一致性，也可以缓解网络通信的压力。

然后研究并设计了可以用于动态图计算的增量式算法。增量式算法的重要特点是在动态图中每次变化部分的计算都可以借助上次计算的结果，实验表明，在数据的变化量在每次 10%-20%的时候，增量式算法相较于非增量式算法在时间效率上大大提升，但是准确率却并没有下降多少。增量式算法的有效性主要体现在增量式算法相较于全量式算法的完整性和收敛性上面。也就是将新增部分对于原有部分上影响使用一个阈值来决定是否进行进一步传播，一旦新增的部分对原有部分的影响小于阈值立即停止传播，从而可以快速的达到收敛。

最后则是基于 HBase，Spark GraphX，Docker，Spring Boot 等技术实现了一个完整可用的图计算平台，这个计算平台主要包括主体存储模块，增量计算模块，动态数据数据导入模块，业务平台展示模块等。

1.4 论文结构安排

本文主要分为六个部分，每个部分按照下列结构进行组织：

第一章，绪论。本章主要有研究背景，国内外研究现状，本文研究内容等部分。研究背景主要介绍了对于本文中研究内容的选题来源以及研究的必要性。国内外研究现状主要描述了国内外的学术界和工业界在这个问题上的最新研究，主要包括图存储系统，图计算系统和动态图计算系统，同时表达了本课题的研究潜力。研究内容着重阐述了本文中的主要研究点以及最后达到的效果，最后的论文组织结构则是对全文的论文组织做了综合概述。

第二章，关键技术。关键结束方面主要介绍了本文中用到的一些前沿研究技术，以及这些技术现在的发展状况。主要包括图存储的的实现方式，分布式图存

储和单机图存储，以及分布式图存储涉及到的图分割技术。图计算方面主要包括图计算的基础理论，分布式图计算和单机式图计算。

第三章，动态图存储的研究与设计。本文的主要研究点之一，动态图相对于静态图多了一个时间维度，因此实现起来的难度大大增加。图的动态存储最重要的是可以获取某个时刻整个网络的快照，在这里主要是利用了特殊的数据结构和划分时间区间的存储来实现。通过 RabbitMQ 系统实现主体存储和计算副本之间的相互通信，保证了主体存储和计算副本之间的一致性。

第四章，分布式图计算系统的研究与设计。本文的主要研究点之一，本文中的分布式图计算系统一个重要的特点就是可以支持增量式计算，也就是支持动态图的分析。通过对现有的全量式算法进行改造，实现了增量式 PageRank，增量式 ShortestPath，增量式 TrustRank 和增量式 TunkRank 算法。并且在这一章的最后做了大量的实验，验证了增量式算法的有效性。

第五章，基于本文的研究成果实现了一个完整的可用的图计算平台。这个图计算平台包括主体存储模块，增量计算模块，动态数据数据导入模块，业务平台展示模块，并且包装了可以直接调用的接口。

第六章，总结与展望。这一章主要总结了前几章的工作成果，做了一个全面的总结。并对前几章中未解决的为题提出了思考，作为下一步待解决的问题。

相关技术介绍

2.1 图存储系统的介绍与研究
第二章

本章的重点是介绍图存储的发展历程及最新成果。先是是对图存储的数据结构、算法以及工业界的实现做了全面具体的梳理。这些问题为本文中动态图存储系统的设计提供了基本理论和优化方法。后一部分是针对于近些年大数据技术的兴起，图计算在这一方面的发展，详细的阐述了图计算的应用场景。最后是总结了现有图计算和存储方面的成果和缺陷，表明本文研究的必要性。

2.1.1 单机图存储

现在图研究领域在图存储方面主要分为两个派别，单机式图存储和分布式图存储。两个派别争论的焦点便是究竟图的存储适不适合分布式存储。首先介绍单机式图存储。

2.1.1.1 单机图存储的依据

在分布式技术成熟以前，长久以来存储技术一向是单机的。特别是图这种结构，由于其复杂的结构特征，它的存储一直是让人们比较复杂的问题。对于图的邻接矩阵存储，邻接表存储和十字链表存储，不论从哪一种结构上来看，分布式的实现代价都不小。而且要想分布式存储，图的最优分割点[27]是一个 NP 难的问题，一旦分割的算法实现的不好，所造成的通信干扰非常之大，严重的影响到了图查询和图计算的效率。

因此支持单机式图存储的人们认为把一个复杂网络分割后存储并不是一个最好的选择，随着硬件技术的革新，完全可以通过增加硬件配置的方式完成一个大规模网络的存储，并且可以保持图的原生特性。

2.1.1.2 典型实现方式

Neo4j 是这派技术的典型代表产品。Neo4j 利用类似于十字链表的方式，将点和边分别采用定长字段存储，每个点存储第一条入边和第一条出边的索引，每条边存储它的首尾节点索引和上一条边与下一条边的索引。每当在 Neo4j 中进行一次关系查询时，都会使用深度优先或者广度优先的方式从查询顶点开始遍历链表，直到找到相应的顶点或者边为止。这样的查询相较于传统的关系型数据库来说查询速度能高一个数量级。

2.1.2 分布式图存储

2.1.2.1 分布式图存储的必要性

随着近些年来分布式技术的逐渐稳定，图分割任务在现有的计算能力下看似也不是一个困难的问题了。虽然单机式图存储可以通过堆积硬件来实现，但是近些年数据的扩展规模越来越快，硬件的革新速度跟不上数据的产生速度。因此图的分布式存储还是一个特别紧迫的需求，特别是在一些海量计算任务上面，分布式存储可以方便的与分布式计算技术结合，将会使处理效率大大提升。

分布式存储的难点是图分割技术，图分割的方式选择将会影响到之后的查询和计算效率。图分割的方式会直接决定之后在其上运行系统的效率。

衡量一个图分割算法好坏主要在三个方面：

1. 通信代价：图计算的主要依据是图遍历，而图遍历会涉及到不同节点之间的通信，加入一个遍历操作的节点分布在多个机器上，不同机器之间的网络通信将会极大地影响计算效率。因为在本机内存上查找的时间可能在纳秒级别，而不同机器间的网络通信会在毫秒级别。
2. 负载均衡：主要是要平均的使用分布式系统上每台机器的计算能力，假如图的分布规模不均匀，那会造成部分机器过载和另一部分机器空载的情况，影响分布式系统计算能力的发挥。
3. 存储冗余：对于一个分布式系统来说，为了保证容错性往往会进行复制以保证高可用。对于分割不好的图也会造成大量的存储冗余，比如分割点在比较重要的节点处。

2.1.2.2 Hash 分割

Hash 分割[28]是最简单的一种分割，这种分割不会去考虑通信代价、负载均衡、存储冗余等问题。比较适合图比较小的时候的分割，因为对图分割也是需要时间消耗的。研究表明在图的节点规模在 10 亿以下时，采用 Hash 分割的方式完全可以满足大部分的需求，当图的节点规模在 10 亿以上时，采用复杂的算法分割效果更优。

2.1.2.3 启发式算法分割

图的分割问题是经典的 NP 完全问题，因此大多数的图分割算法都是基于启发式算法来实现的。主要分两种实现方式：一种是启发式的交换点对，另一种是多层次的划分。

启发式方法的代表是 KL[29]和 FM 法。对于图分割的问题，早在上世纪的八十年代便有了深入的研究，主要是科宁汉（Kernighan）和林（Lin）[31]提出的一种非常有效的启发式算法叫做 KL 算法。KL 算法的主要思想是在每轮的划分中都把一个图随机的进行 2 等分，然后交换两部分的节点，分别计算交换了点之后得到的收益，再把收益率最高的节点进行再次对换。该算法的最坏时间复杂度为 O(n^2)，对于复杂的图分割问题来讲这个时间复杂度是可以忍受的。但是这种方

法也有其缺点，他最多只能处理 10 的 4 次方以内的节点数量，再高规模的节点数量效果就有些不好了。

为了处理大规模的图切分任务，库玛（Kumar）等人之后提出了分层图划分算法 METIS[32]。METIS 算法的核心是先将大图进行一定方法的“粗糙化”，再对粗糙化之后的每个小图执行 KL 算法，这样的改进成功的克服了 KL 算法无法支持大规模数据划分的缺点。百万级的图分割问题基本上可以在秒级时间代价内完成。

2.2 图计算系统的介绍

2.2.1 图计算基础理论

随着互联网迅速的发展，大规模图分析的需求也越来越紧迫。因此近些年涌现出来了一批图计算系统，这其中具有标志性意义的有 Pregel，GraphLab，VENUS[33]和 GraphChi[34]等。他们实现了类似于 Hadoop 和 Spark 这样的框架，通过简单的更新函数就可以完成大规模图计算。现有的图计算的基本思想是使用“Think like a vertex”去表达图计算的过程，使得庞大的计算任务可以转换成顶点任务就可以完成。

顶点任务程序只用计算更新图中顶点和边的更新状态，然后传播到周围的节点。图计算中有两个重要的计算机制，一个是 BSP 整体同步计算模型[35]，另一个是 GraphLab 的异步并行计算模型。

具体的来说就是，图计算系统会预先定义好一个顶点更新函数 compute，用户要想使用该系统做自己的计算任务，只需要重载该函数即可。compute 函数可以使用节点的信息进行计算，也可以使用邻接边的信息。每次迭代过程就是每个节点都执行自己的 compute 函数。

2.2.1.1 BSP 算法

BSP（Bulk Synchronous Parallel，整体同步计算模型）[35]是一种新兴的分布式计算模型，是由哈佛大学的 Viliant 和牛津大学的 Bill Coll 提出的并行计算模型，最开始被称为“大同步模型”，又因为他们提出这个模型的初衷是改变现有的计算机体系结构，用来作为计算机体系结构和计算机语言之间的沟通桥梁，因此它还有个别名“桥模型”。

BSP 由一系列的全局超步计算过程组成（每个全局超步就是一次迭代过程），每个超步主要进行三个任务：

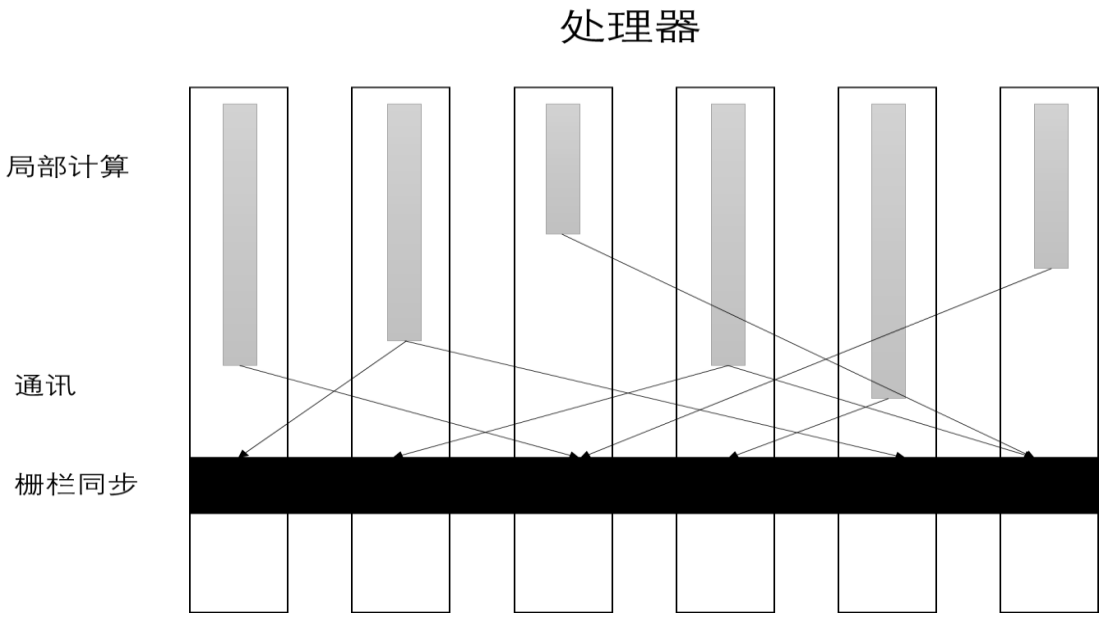
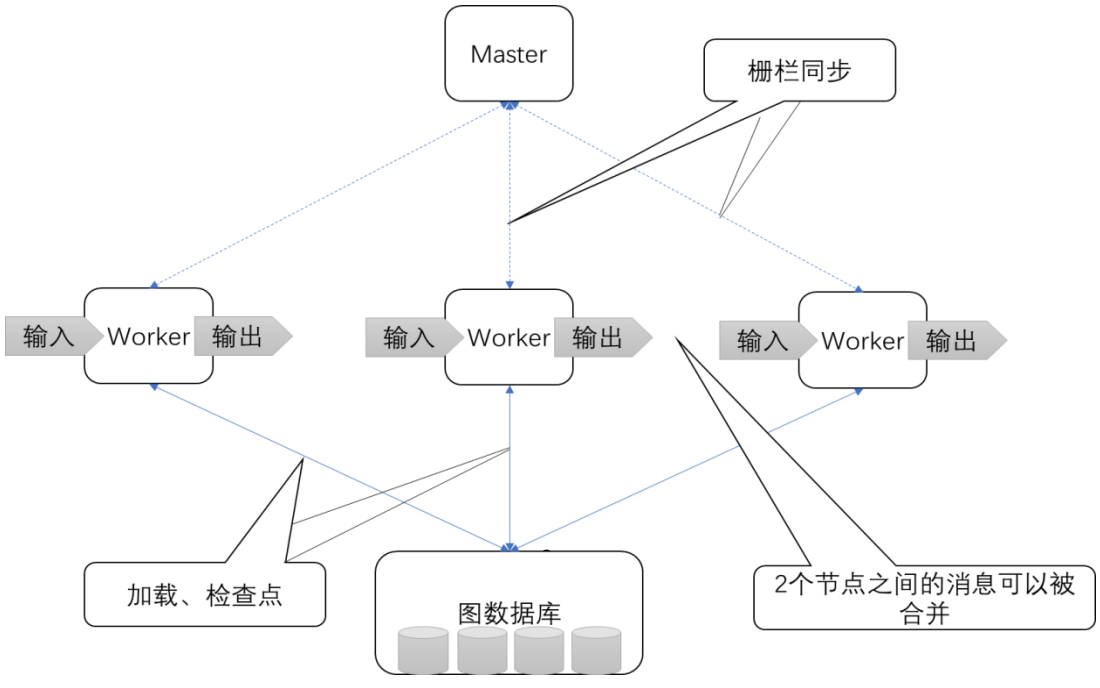


图 2-1 BSP 计算模型

1. 局部计算：每个 processer 都只计算自身的计算任务，他们只读取存储在本地内存中的值，不同的计算任务之间是相互独立的并且异步执行。
2. 通讯：每个 processor 计算完自身的任务之后，会将消息传递到与他相关联的 processor。
3. 栅栏同步：所有的 processor 都处理完毕并且消息交换完成以后，会进行下一个超步，也就是下一轮迭代。

2.2.1.2 Pregel 算法

Pregel 最早是由 Google 提出的大规模图计算模型，现在已被大量的新兴图计算框架所采纳。Pregel 是基于 BSP 实现的并行图处理系统，可以运行在多台廉价计算机组成的集群上面。通常一个图计算任务会被分解到多台计算机上同时执行，任务执行的过程中，临时数据一般会存储在本地磁盘，而持久化数据会存储



在分布式文件系统。

图 2-2 Pregel 体系结构

Pregel 的体系结构如图 2-2 所示：

- 集群中多台机器一起执行计算任务，其中一台机器作为 Master，其他的机器作为 Worker。
- Master 负责着管理其他的 Worker，与 Worker 进行通信。每当一个计算任务来临 Master 将图分为多个分区，并且将分区分配给每个 Worker，监控 Worker 的运行。
- Worker 会向 Master 注册自己的信息。每个 Worker 管辖自己分区的内存状态，计算状态被保存在自身内存中。每个超步中每个 Worker 都会遍历自己内存中的节点，并且计算每个节点的 update 函数。
- Master 使用定期发送消息的方式进行容错。Master 定期向 Worker 发送消息，一旦消息在一定的时间内没有的到响应，Master 会认为这个计算节点出现了错误，然后会启动恢复模式。

Pregel 顶点间的信息传递采用纯消息传递的模式，在使用异步和批量消息传输的方式下，可以极大地提升整个系统的性能。

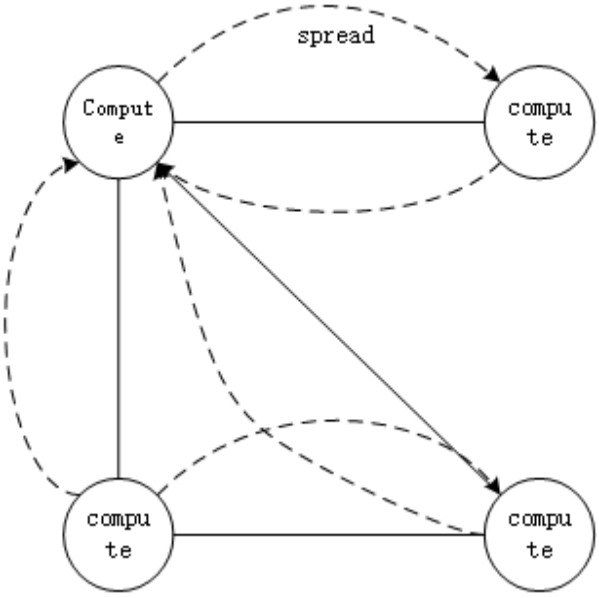
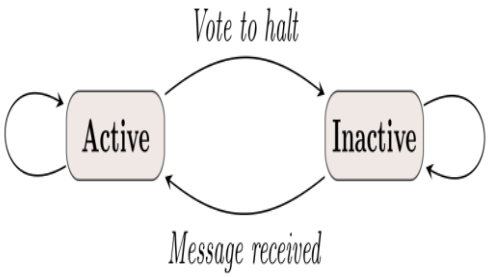


图 2-3 消息传递模式图



2-4 状态机图

图 2-3 是 Pregel 的消息传递模式图，该函数描述了一个顶点在一个超步 S 需要进行的操作。该函数读取前一个超步（S-1）发送过来的消息，在这个节点上做聚合，聚合以后修改本身的信息，然后沿着出边发送自身的信息。这些在超步 S 发送的信息会在（S+1）的超步被别的节点接收。

图 2-4 是 Pregel 执行的状态机图，每个收到消息的节点都会被设置为活跃，活跃的节点可以参与本次计算并且发送消息，非活跃的节点如果收到消息就会变为活跃。迭代算法执行到图中所有的节点都变为非活跃，就可以结束。

2.2.2 单机图计算系统

单机图计算系统的代表是华为的 VENUS 和伯克利大学的 GraphChi。VENUS 提出了以顶点为中心的流水化图计算模型，这样的做法显著地降低了基于磁盘的图 IO，极大地改善了磁盘 IO 与计算速度不匹配的情况。VENUS 使用了独有的基于外存算法的存储扩展方式，叫做分级存储的基于流水线的图计算模型，可以突破单机图计算的可扩展能力，达到了不错的计算效果。

VENUS 系统使用了数据分片的方法以及独有的外存模型。在该系统中，每个数据分片被分为 v-shard 和 g-shard 两个部分。V-shard 用来存储图中的点部分，数量不大，并且需要频繁的修改。G-shard 用来存储图中的边数据部分，数据量巨大，但是几乎不发生改变。在图计算的执行过程中，首先磁盘按照顺序读取 g-shard 分片的数据，每当遇到 g-shard 所关联到的点的时候，就将 v-shard 中的点也加载到内存。处理 g-shard 中边对应的更新函数，此时 g-shard 关联的点已经在内存中了，可以直接读取计算。一旦计算完成，立马丢弃 g-shard 中加载过来的边数据，这样可以保证系统在加载数据的同时进行计算，极大地减少了运行的时间。

另一个基于单机的图计算系统是 GraphChi，在 GraphChi 中一个图也会被分割成多个数据片，保证每个数据片都可以放在内存里面。每个数据片内存存储的都是一个子图，包括一系列的顶点和边。图计算过程由多个迭代过程组成，每个迭代过程都会对图中的节点进行更新计算。在 GraphChi 中，每次迭代系统都会一次的处理一个数据分片，并且在每个数据分片上执行更新函数。这部分的处理主要分为单个具体的步骤。首先是载入，将一个数据分片载入到内存中。然后是更新，在每个数据分片中的每个节点上执行更新操作。最后是写回，将计算好的信息重新写会到磁盘上。这种处理方法的缺点主要有二，一个是在数据加载的过程中将会产生很大的 IO，而这部分 IO 的过程中无法同时进行计算。第二个是每个分片计算完后会传播本身的信息给相邻分片的节点，这样将会产生大量的随机 IO。

2.2.3 分布式图计算系统

2.2.3.1 GraphLab

GraphLab 是一个基于内存的分布式图计算系统，在 GraphLab 中一个图被分割成多个子图，每个子图被存储在一台单独计算机的内存上，图分割采用的是顶点切分的方式。图分割之后，被切分的顶点会在每个与他有关联的自图中都存在

一个备份。这样一个顶点就可能会存储在多个主机上，每次在一个子图上发生计算，更新过信息的节点就会将计算结果同步到其他有该节点备份的主机上。图上的数据包括图的顶点和边，以及顶点和边上的值。这样做的好处就是不用网络传输图结构信息了，但是同步节点信息仍然会需要大量的网络通信。

2.2.3.2 Spark GraphX

Spark GraphX 是基于 Spark 平台的一个分布式图计算系统，随着 Spark 的开源影响力的进一步增加，Spark GraphX 也逐渐的被更多的人熟知。

Spark GraphX 以 Spark 的内存结构 RDD 作为存储结构，并且开放出了 VertexRDD 和 EdgeRDD 两种 API，方便用户进行图操作。VertexRDD 和 EdgeRDD 采用不同的分区方式，VertexRDD 一般情况下采用的都是 Hash 划分的，因为在绝大部分的图中 Vertex 的个数都是远小于 Edge 的。而 Edge 采用指定的划分算法进行划分。Spark GraphX 还开放了将 VertexRDD 与 EdgeRDD 结合的 triplets 方法，这在执行一些沿着边传播的算法的时候非常方便。对于 MapReduce 的支持方面，Spark GraphX 开放出了 aggregateMessages 函数，该函数有两个重要的操作，一个是 sendMsg 方法，另一个是 mergeMessage 方法。Spark GraphX 的出现，使我们可以仅仅调用一些简单的 API 就可以实现复杂的图计算任务。

但是 Spark GraphX 也并非图计算的最终形式，在图的分割，消息的传递，图的动态式计算方面仍然有很多可以提高的点。

2.3 本章小结

本章主要介绍了本课题中相关的关键技术。由于本课题中要研究实现一个图计算系统，涉及到的技术主要分为图的分布式存储和分布式计算。第一节介绍了现有的图存储方式以及他们之间的优劣关系，为下一章文中提出的存储结构提供依据。第二节主要介绍了现有的图计算技术，以及他们各自适用的场景，分析了他们的适用情况以及存在的不足，为本文的图计算系统做出了铺垫。

动态图存储系统的研究与实现

虽然目前关于图存储的研究目前已经有比较成熟的体系了，但是关于动态图的存储第三章重要的研究成果。主要是因为动态图的存储涉及到具体的分析行为，很难做出一个统一的规范。本章根据动态图计算系统的需求，研究并设计了一个可以快速获取动态图的某一时刻静态图快照，并且进行计算的动态图存储系统。该系统主要分为主体存储模块和计算副本模块。

3.1 存储系统整体架构

动态图存储架构如图 3-1 所示，主要分为两个模块。第一个是主体存储模块，主体存储模块主要是为了完成图上常用的增、删、改、查等操作。主体存储模块可以细分为两个子模块，子模块一叫做磁盘存储模块，这部分主要负责图的持久化。另一部分叫做中间存储，主要封装了一系列的图操作过程，对应图在内存中的存储模型，同时也封装了时序图的操作逻辑。

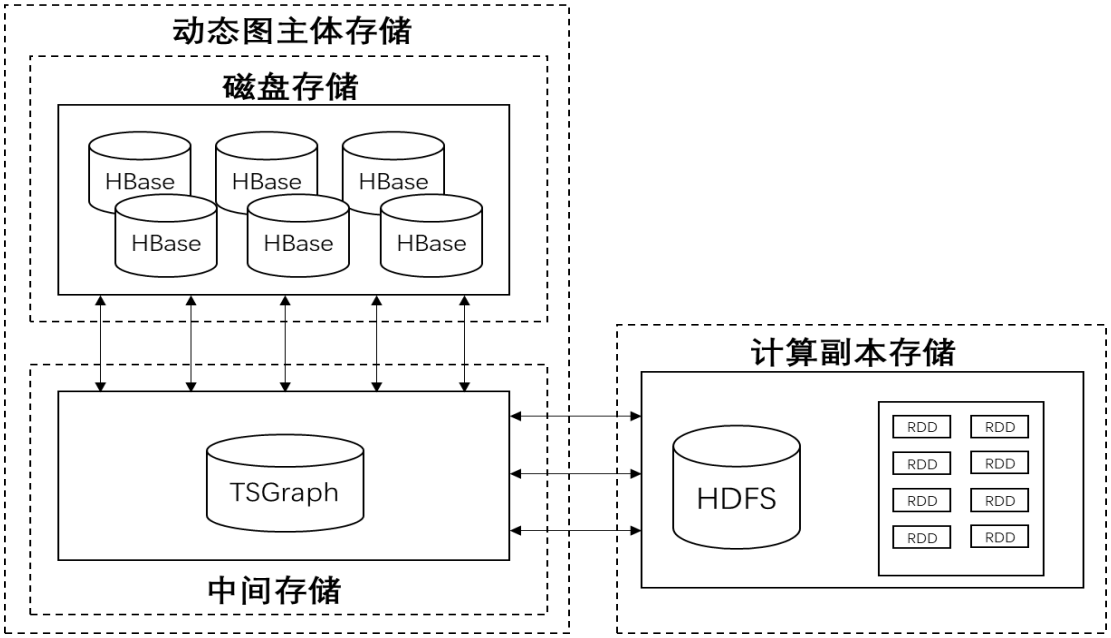


图 3-1 动态图存储架构

3.2 主体存储的研究与设计

在图处理领域一个重要的目标是实现图关系的快速查询，现实中的主要场景有在社交网络中查询某个人的朋友，查询某两个人的共同朋友，推荐可能认识的朋友，查询任意两个人怎么样才能快速认识。在科研论文方面查询引用的论文，

有共同作者的论文，有共同关键词的论文。互联网网页中一个页面链出的页面，链入的页面等。这些需求的共同点是查询的准确性和快速性，这就要求我们的存储系统可以快速的响应查询，同时还要有一定的安全性。

3.2.1 图存储结构的研究

图的存储方式主要有邻接矩阵、邻接表和十字链表等。邻接矩阵存储稀疏图会造成巨大的浪费，单一邻接表无法做到双向遍历，不能满足一些复杂查询。十字链表可以双向遍历但是在范围查询方面表现的并不是很好，无法满足动态图存储中的按时间区间检索。因此，在本文中采用十字链表作为内存结构，用来存储动态图按时间检索生成的静态图结构。动态图的存储采用另一种灵活的邻接表存储方式。同时为了快速查询，采用 BigTable[36]模型作为存储后端。

3.2.1.1 内存存储结构的研究

内存里最重要的是进行图遍历操作，图遍历操作的基础是深度优先搜索和广度优先搜索。因此快速的访问一个节点的上下游节点是内存存储首先要解决的问题。根据图存储的几种模式的特点，邻接矩阵虽然可以做到快速的深度优先搜索和广度优先搜索，但是占的内存较大，在内存比较宝贵的情况下是肯定不合适的。在这里使用改进的十字链表存储方式。

具体的改进思路是将节点（Node）和边（Edge）都设计一种特殊的数据结构，用定长字段表示，节省存储空间。节点 id 作为索引，便于快速查找。Node 的结构如图 3-2 上方所示，为了节省空间只存储节点 id，下一条关系 id（nextRelId），下一条属性的 id（nextPropId）。nextRealId 指向下一条关系，nextPropId 指向第一个属性。Edge 的存储结构如图 3-2 下方所示，主要有关系 id，指向头节点的指针（firstNode），指向尾节点的指针（secondNode），关系类型（relationshipType），头节点向前一条关系的 id（firstPrevRelId），头节点指向下一条关系的 id（firstNextRelId），尾节点向前一条关系的 id（secondPrevRelId），尾节点指向下一条关系的 id（secondNextRelId），第一个属性 id（nextPropId）。

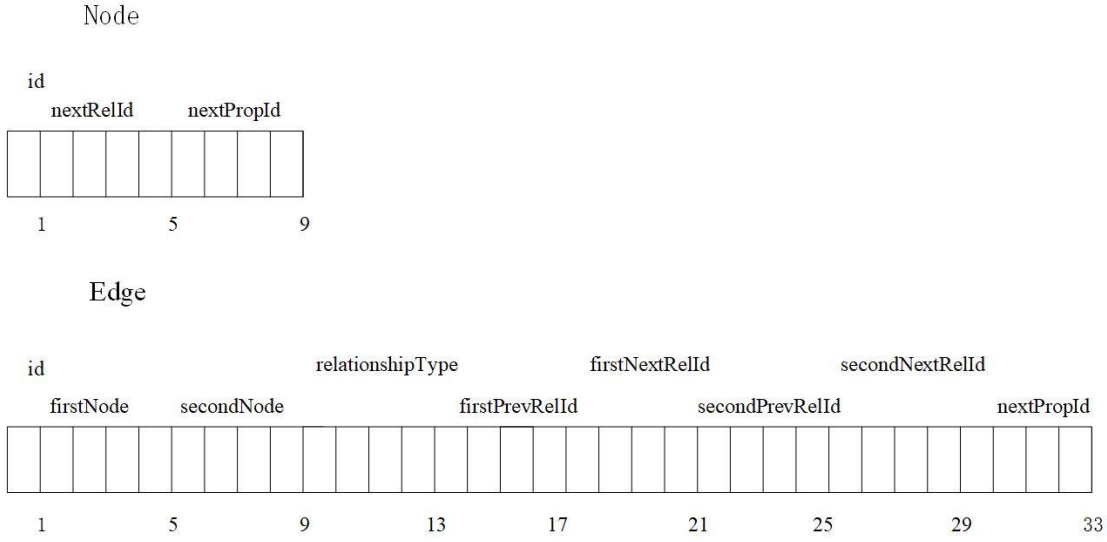


图 3-2 内存存储结构

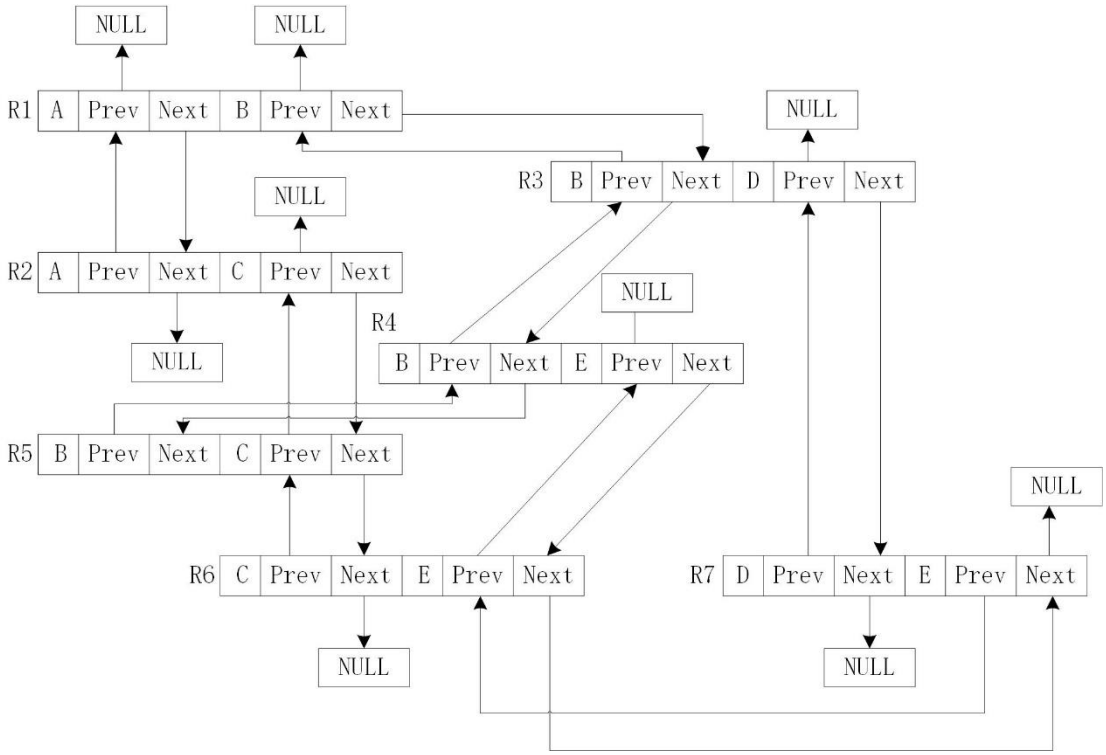


图 3-3 内存遍历过程

图 3-3 是使用该存储模型在内存中的遍历过程。如果要遍历图中节点 B 的所有关系，只需要按照节点 B 的指针 next 方向一直遍历即可，直到指针指向 NULL。从图中可以看出节点 B 的所有关系为 R1、R3、R4、R5。

3.2.1.2 磁盘存储结构的研究

为了满足快速检索以及分布式存储的特性，经过多次实验对比，我们采用 BigTable 作为底层存储模型。在 BagTable 的数据模型下，每个表都是行的集合，每一行都由一个唯一主键标识。每行由任意有限大小的单元（cell）组成，每个单元由一个列（column）和值（value）组成。在 BagTable 存储模型下，每行都支持大量的单元格，并且这些单元格不用像在关系数据库中那样需要预先定义。

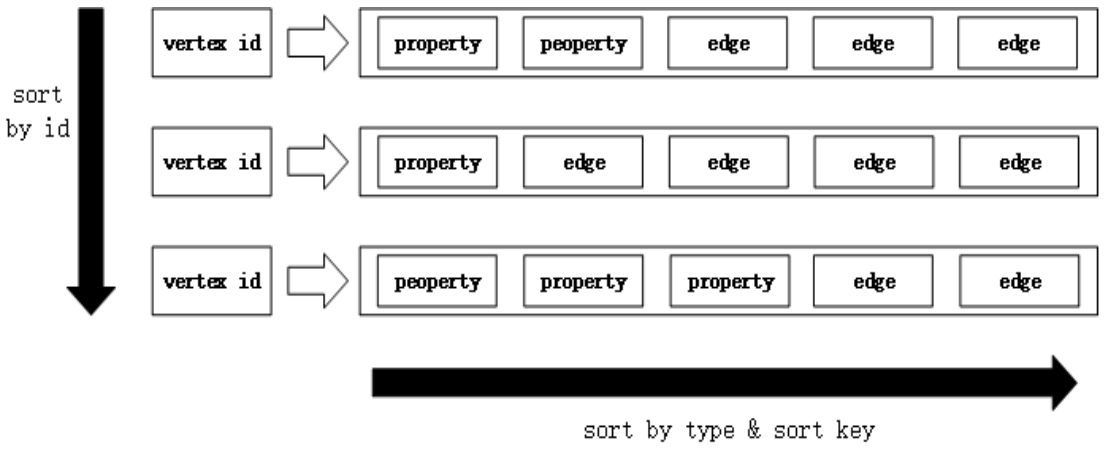


图 3-4BigTable 图存储模型

如图 3-4 所示，将每个节点的邻接表作为一行存储在 HBase 中。顶点 id 是该顶点邻接表的行关键字，每条边（包括出边和入边）和属性都作为一个独立的单元存储在行中，这样可以高效的删除和插入。特别是在使用 vertex id 作为关键字之后，去掉可以将该表按照 id 进行排序，从而在图切分的时候，将经常互相访问的节点存储在同一个分区中，可以极大地加快访问速度。

每一行的数据可以按照某个 type，或者是按照自定义的 sort key 排序，可以加速查询某个节点的邻接关系时的速度。

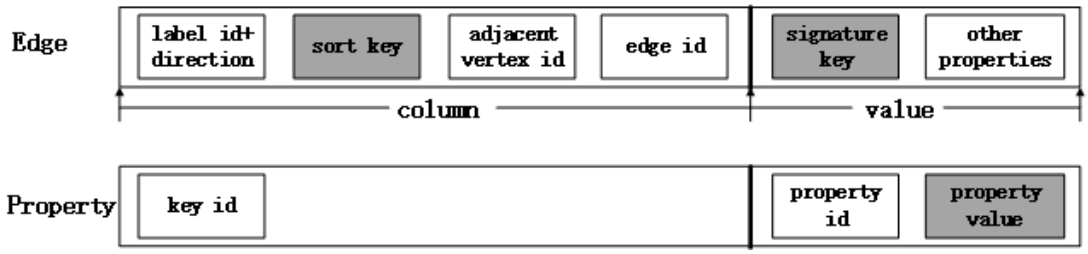


图 3-5 单元存储模型

边（Edge）和属性（Property）的存储结构如图 3-5 所示。每个顶点的邻接边和属性都作为一个单独的单元存储在以这个顶点 id 为键的行中。每个单元都被划分为两部分，column 和 value。在 Edge 的存储单元中，column 是由 label id&direction，sort key，adjacent vertex id，edge id 等四部分组成。label id&direction 表示边类型以及方向，sort key 表示排序字段，adjacent vertex id 表示邻接节点的 id，edge id 代表边本身的 id。

由于每个边和属性都作为一个单独单元存储在对应顶点的行中。序列化的时候保证 Edge 和 Property 按照一定的顺序排序，并且使用可变 ID 编码方案和压缩对象序列化来保证每个边或属性占用的存储单元尽可能小。如图中浅色框部分表示支持可变长编码方案的存储区域，这部分经过一定的编码处理可以有效的减小他们消耗的字节数。深色部分框部分表示使用关联属性表中压缩数据元序列化的属性值（即对象）。

3.2.1.3 动态图存储的研究

为了满足实时数据的分析，近些年来涌现了很多时序数据库。比较有名的有 InfluxDB[37]，RRDtool[38]，Graphite[39,40,41]，openTSDB[42]等。InfluxDB 和 RRDtool 都是老牌的单机时序数据库，Graphite 是基于 whisper[43]开发的时序数据库，较 RRDtool 做了些优化。openTSDB 是一个分布式，可伸缩的时序数据库，支持较高的吐出量，存储峰值可达每秒百万级别，并且支持时间精度到毫秒级别的数据存储。但是针对图数据的存储，至今仍然没有一个统一的方案。主要是图数据的存储和传统时序数据库的存储目标有较大差别。传统的时序数据库只关注一段时

间内的数据，分析主要也是针对阶段时间数据做分析，比如传感器获取的实时数据，摄像头采集的实时数据等。但是在图的模型中，需要全量的数据才能反映整个网络的状态，大图中某个区域的节点和边的变化影响的是整个图的属性。因此可以采用图快照的方式反映大规模网络演化过程中的某个时刻的静态图。

首先按照传统的图表示方式，不含时序信息的静态图记为有序对：$G = (V, E)$。记这样的图为静态图，其中 V 表示顶点（Vertices 或 Nodes）的集合，E 表示边（Edges 或 Links）的集合。这里的 E 中的一个元素 e 是 V 中的两个元素 u，v 组成的二元组$e = (u, v)$。在有向图中 u 为源点，v 为目标顶点。

对于有时序信息的动态图，可以相应的记为：

$$G(t) = (V(t), E(t))$$（3-1）

这里的 t 代表的是时序信息，$V(t)$和$E(t)$则分别代表了具有时序信息的顶点和边。动态图的时序信息表示的是该结构或者该属性值的生命周期，可以使用非重合时间区间的集合来表示。一个完整周期的动态图我们用$G = \{G_{t1},\ G_{t2} \ldots G_{tn}\}$表示，任意时刻的静态图即为$G_{tn}$。

3.2.2 主体存储的设计与实现

3.2.2.1 TSGraph 的设计与实现

TSGraph 作为中间存储结构主要负责和磁盘存储的交互，完成图在内存中的增删改查和按照时序检索。同时 TSGraph 也是磁盘存储和计算副本沟通的桥梁，TSGraph 会按照时序查询得到动态图的某个静态快照，并将这个静态快照发送到计算副本上供计算副本计算。同时计算副本完成计算以后，也会把该静态图的计算结果返回到 TSGraph 中，TSGraph 负责将计算好的数据与磁盘中的原始数据进行合并，并且在磁盘中存储最终数据。

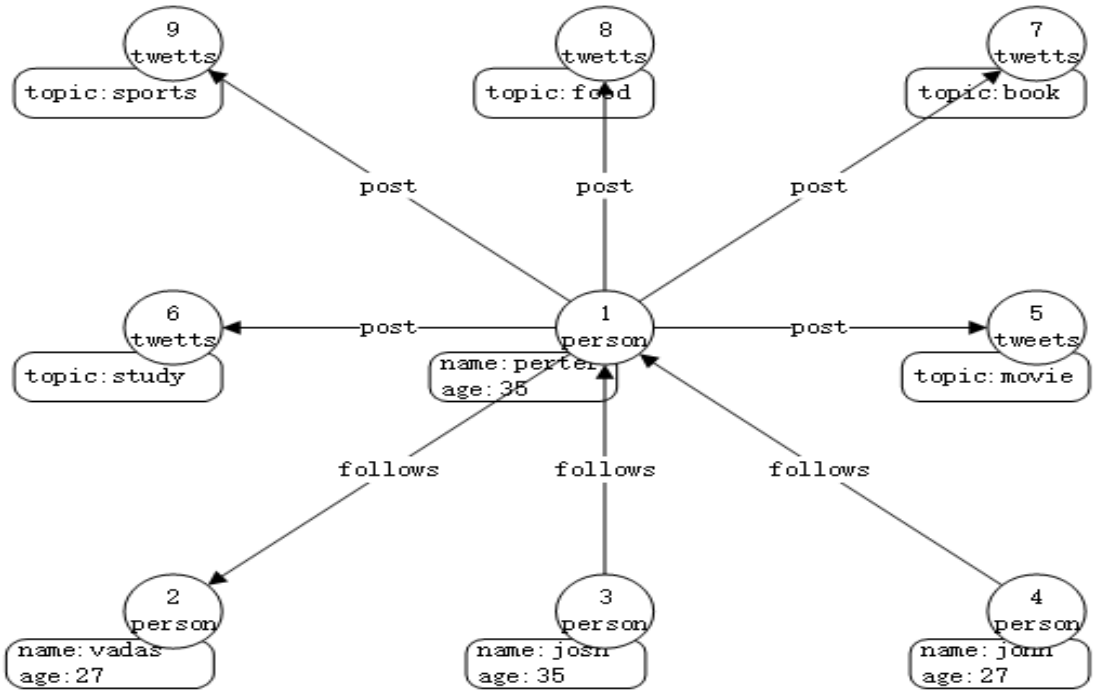


图 3-6 社交关系图存储模型

如图 3-6 所示，一个图可以抽象为节点（vertex）和边（edge）的集合。每个点都有自己的标签（label）和属性（property），每个边也具有自己的标签（label）和属性（property）。

一个大图由一系列的节点和边组成，具体组成如下所示。

1. 点的集合
   - 每个节点都有一个唯一的标识。
   - 每个节点都有一个出边的集合。
   - 每个节点都有一个入边的集合。
   - 每个节点一个由键值对组成的属性的集合。
2. 边的集合
   - 每条边都有一个唯一的标识。
   - 每条边都有一个尾部节点 id。
   - 每条边都有一个头部节点 id。
   - 每条边都有一个由键值对组成的属性的集合。

根据以上描述我们设计出几个重要的类，如类图 3-7 所示，主要包括 GraphFactory 类，用来连接存储后端。Graph 类，图上一系列操作的集合。Edge 类，边属性和行为的集合。Vertex 类，节点属性和行为的集合。Property，节点和边的属性。GraphIndex 类，索引行为的集合。

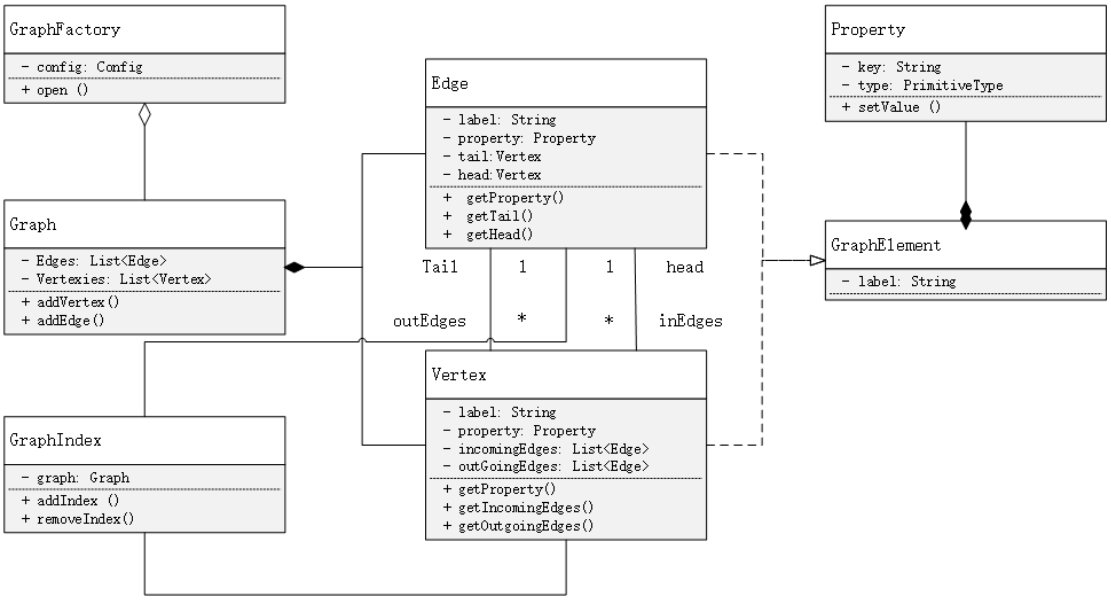


图 3-7 类图

GraphFactory 用来建立与后端存储的连接。主要是根据键值对的配置来确定需要实例化的图。GraphFactory 有 build，open，close 等主要方法。build 方法用

于使用配置操作实例化一个图。open 方法用于使用传入配置实例的方法实例化一个图。close 方法用于销毁这个实例化的图。

Graph 类是图上一系列操作的集合。主要包括 traversal，addVertex，addEdge，removeVertex，removeEdge 等方法。teaversal 是获取 edge 和 vertex 的实例，用于执行图上的遍历。addVertex 用于增加节点，addEdge 用于增加边，removeVertex 用于删除节点，removeEdge 用于删除边。

Edge 类是图上边以及边行为的集合。主要包括 edgeLabel，inVertex，outVertex，otherVertex，property，addPropertey，removeProperty 等方法。edgeLabel 用来获取这条边上的标签类型，inVertex 用来获取这条边起始的节点，outVertex 用来获取这条边尾部的节点，otherVertex 用来获取边上另一侧的节点，property 用来获取边上的属性。

Vertex 类是图上的节点以及节点行为的集合。主要包括 vertexLabel，property，addProperty，removeProperty，addEdge，removeEdge，edges，inEdges，outEdges 等方法。vertexLabel 用来获取节点的标签类型，addEdge 用来给两个节点之间增加一条边，removeEdge 用来删除两个节点之间的边，edges 用来获取一个节点周围所有的边，inedges 用来获取一个节点的入边集合，outEdges 返回节点的出边集合。

Property 类是节点和边属性及行为的集合。主要有 key，propertykey，element 等方法。

3.2.2.2 后端存储的设计与实现

对于时序图数据的存储来说最重要的是要获取动态图的某一时刻快照，为此需要给图中的节点和边数据都加上时间(timestamp)属性，用来代表数据的版本。在这里采用 HBase 作为存储后端。HBase 是一个构建在 HDFS 上的分布式存储系统，是基于 Google Bigtable 模型开发的典型的 key value 存储系统。HBase 设计的初衷就是实现 HDFS 上的海量数据的存储，是一个很适合存储非结构化海量数据存储的数据库，主要特点是它是基于列的存储，而不是基于行的存储，这样的存储结构可以方便读取海量数据的内容。HBase 的基本存储结构就是类似于哈希表的概念，但又不仅仅是简单的一个 key 对应一个 value。每个 key 对应的可能是多个属性的数据结构，但是又没有传统数据库中那么多的关联关系，也可以叫做是松散数据。

在 HBase 中存储的数据可以看做是一张巨大的表，这个表的属性是可以根据我们的需求动态增加的，在 HBase 中没有表与表之间的关联查询。在存储数据的时候只要制定了要存储数据的 column 和 family 即可，不用关心数据的类型，在 HBase 中所有的数据都默认为字符的存储。HBase 存在着如下的特点：

- HBase 可以存储的数据规模非常大，一个表可以由数十亿行和上百万列；
- 每行的数据都有一个可排序的主键和任意多的列，并且列的数量是可以根据需求动态增加的，同一张表中的不同数据可以有完全不同的列。
- 面向列的存储和控制，面向列的独立检索；
- 空列并不会占用存储单元，这样可以让我们的表设计的非常稀疏
- 每个单元中可以存储多版本的数据，在默认情况下版本号是自动分配的，是插入的时间戳，并且可以由用户定义。
- 数据类型单一，HBase 中存储的数据都是字符串的形式，并不区分他们的具体类型。

因为 Hbase 的每个单元内的存储支持多版本存储，每个版本的版本号用时间戳表示，经过一定的修改完全可以快速的查询到某个时间段的静态图。Hbase 中一个数据多版本的表示方法如下：

```
COLUMN                          CELL
 v:c1                           timestamp=1499088390024,value=value7
 v:c1                           timestamp=1499088387559,value=value6
 v:c1                           timestamp=1499088385347,value=value5
 v:c1                           timestamp=1499088383228,value=value4
 v:c1                           timestamp=1499088380943,value=value3
```

在 Hbase 中可以设计存储多个版本的数据，版本个数和版本号表示方式都可以自己定义，如上图所示，c1 的值有 5 个版本，每个版本都对应了唯一的 timestamp。查询的时候可以具体到某个版本的数据，对应到图存储中，将每个时间区间的静态图存储为同一个版本，获取某个时间区间的静态图用版本号查询就可以做到。

图 3-5 的数据用关系型数据库存储可以转化为这样的模型，图中有 person 和 tweets 两种节点，并且有 post 和 follow 两种关系。Person 和 tweets 节点可以笨别转化成一个 person 表和一个 tweets 表，post 关系可以将 tweets 表的外键指向 person 的主键 id。Follow 关系可以用另一个 follow 表表示。

表 3-1 tweets 表

tweets		
id	topic	person
1	study	1
2	sports	1
3	food	1
4	book	1
5	movie	1

表 3-2 person 表

person		
id	name	age
1	peter	35
2	vadas	27
3	john	27
4	josh	35

表 3-3 follow 表

follow		
id	person1	person2
1	1	2
2	3	1
3	4	1

将这个数据转化到 BigTable 的存储模型上，可以将以上三张表转化为一张表，将该数据转化成图 3-3 和图 3-4 描述的存储模型。整个表以 vertex 为 row key，并且排序，然后将经常访问的 vertex 存储到同一个 region 中。Family 有 edge 和 property 两种。存储单元有 edge 和 property 两种。把上述数据转化为 BigTable 存储模型的 Schema 为下表所示。

表 3-4 图 BigTable 模型 Schema

row key	family	Attributes
vertexid	edge	{columns:value} 3versions
	property	{columns:value} 3versions

以图 3-5 中 person 节点中的 josh 作为例子，写出该数据的存储模型。

表 3-5 时序图 BigTable 存储

rowkey	Edge:edge1				Property:property1
	direction	sortkey	adjacentvertexid	Edgeid	Key id
3	out @time1 out @time2	001 @time1 001 @time2	003 @time1 003 @time2	001 @time1 001 @time2	{name:josh,age:35} @time1 {name:josh,age:35} @time2

在查询的时候需要指定数据的 version number。version 可以由用户自己定义，有两种定义 version 的方式，第一种是保存数据的 n 个版本，二是保存近一段时

间内的版本。对于图模型来说，每增加或者修改节点或者边，相应的修改节点和边的版本号为最新版本。每删除一个节点或者边就不更新版本号，当我们查询某个版本节点或者边不存在的时候有两种情况，一种情况是该节点或者边在当前版本被删除了，另一种情况是该节点或者边从来就不存在过，我们要注意这两种情况的分别。

假如在 time3 删除了 josh 到 perter 的 follow 关系。此时 josh 的存储结构更新为下表所示，property 随着版本更新到了 time3，但是 edge 并没有相应的更新，表示在 time3 时刻，jsoh 指向 perter 的边被删除。

表 3-4 时序图存储删除数据的表示

rowkey	Edge:edge1				Property:property1
	direction	sortkey	adjacentvertexid	Edgeid	Key id
3	out @time1 out @time2	001 @time1 001 @time2	003 @time1 003 @time2	001 @time1 001 @time2	{name:josh,age:35} @time1 {name:josh,age:35} @time2 {name:josh,age:35} @time3

查询任以时刻的节点信息使用如下语句：

```
>> get 'twitterGraph','3', {Column => ['edge:edge1:direction'],
   version = time2}
>> COLUMN                          CELL
   edge:edge1:direction            version=time2, value=out
```

3.2.3 计算快照的生成

在进行图计算的时候，如果直接在后端存储上进行计算会造成多种问题。比如如果一个计算的计算过程持续时间过长，并且可能会多次访问图的存储结构，如果正在参与计算的节点或者边此时发生了更新，就会出现不一致的状态。为了解决这个问题，最简单的方式就是在正在进行计算的数据上加锁，直到计算任务完成才释放锁。但是这样又造成了新的问题，系统的吞吐量被严重限制，所以计算与更新共用同一个存储结构并不是一个好的策略。

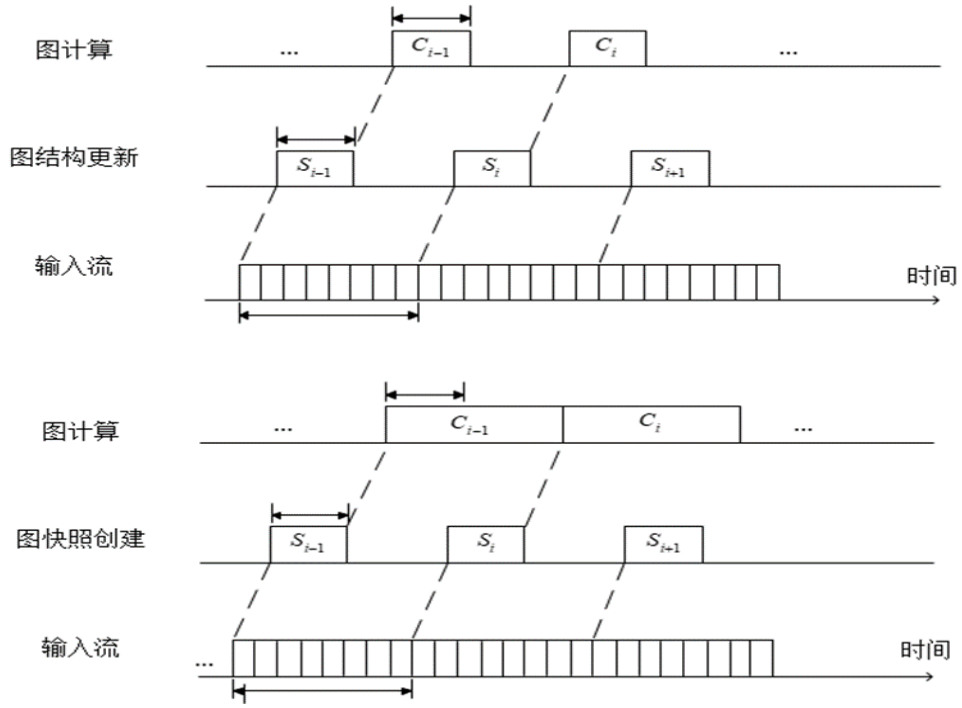


图 3-8 图快照创建与图更新模式的对比

为了解决上述问题，可以将某个时间点的快照拿出来用于分析计算，剩下的部分仍然可以继续进行图更新操作。这样图更新和图计算的所操作的就是不同的版本了，因此也不会造成冲突。而且图更新和图计算负责的主要任务不同，图更新主要执行可以改变图结构的操作，例如增加边，增加节点。而图计算则可以更新图的应用信息。这样的高效隔离使系统有了更高的鲁棒性。

基于以上两点，下一步就可以分析图快照的生成了。生成动态图某个节点的快照我们采用时段提交和累加区间提交两种方式。两种方式分别对应不同性质的任务。两种提交方式都要依赖 Transcation（事务）日志和 Trigger（触发器）技术。

时段提交需要设定好时间区间，设一个记录全局的上一次生成快照的时间 Tlasttime，设一个阶段提交时间区间 Tperiod，每次提交事务的时候都获取当前时间，并且做检查$T_{diff} = T_{present} - T_{lasttime}$。如果$T_{diff} > T_{period}$则生成快照 Si。

```
Begin Transaction{
    T_period = Date.timestamp
    T_diff = T_present - T_lasttime
    if(T_diff > T_period){
    GenerateSubgraph(vertex.timestamp > T_diff && vertex.timestamp <= T_present)
    T_lasttime = T_present
    }
}
End Transcation
```

累加区间提交则是以数据规模来决定计算频度的，适用于实时性稍低的一些任务，节省计算资源。累加区间提交将会设一个记录全局事务 id 的 TransCount，每次更改都会使 TransCount 加 1，并且设置当前事务操作的 TransCount 的值作为节点和边的属性，同时还会有一个$TransCount_{lasttime}$记录上一次生成数据快照的事务 id，每当执行设定值 N 个事务之后，生成一次数据快照。

```
Begin Transaction{
    TransCount = TransCount + 1
    TransCount_diff = TransCount - TransCount_lasttime
    If(TransCount_diff >= n){
    GenerateSubgraph(vertex.transId > TransCount_lasttime && vertex.transId <=
 TransCount)
    TransCount_lasttime = TransCount
    }
}
```

通过这两种快照生成方式，可以保证图的更新和图的计算在两套系统中执行，大大减小了阻塞发生的概率。生成快照的本质就是获取动态图的某个时间段内的静态图。根据 3.1.2 节可知某个时间区间内的静态图获取可以转化为获取 HBase 上某个版本数据的集合。因此计算快照的获取可以转化为对 Hbase 上存储的数据进行按版本查询。在 HBase 上的查询只有按照某一固定的 rowkey 查询和按照 rowkey 的范围查询和全表扫描三种方式。要获取某个版本的数据集合，只能进行全表扫描，这在数据版本非常多的时候效率十分低下。

HBase 具有二级索引的机制，因此可以在 rowkey 与版本的对应关系上建立二级索引。二级索引建立的本质是建立各列值与 rowkey 之间的关系，在这里使用 version-rowkey 来建立二级索引。建立好二级索引的数据存储形式如下表所示。

表 3-7 时序图存储 rowkey 优化设计

Rowkey	ColumnFamily:Index	ColumnFamily:Edge	ColumnFamily:Property
version1-rowkey1	/		
version1-rowkey2	/		
version1-rowkey3	/		
version2-rowkey	/		

1			
rowkey1		edge1	property1
rowkey2		edge1	property1
rowkey3		edge1	property1

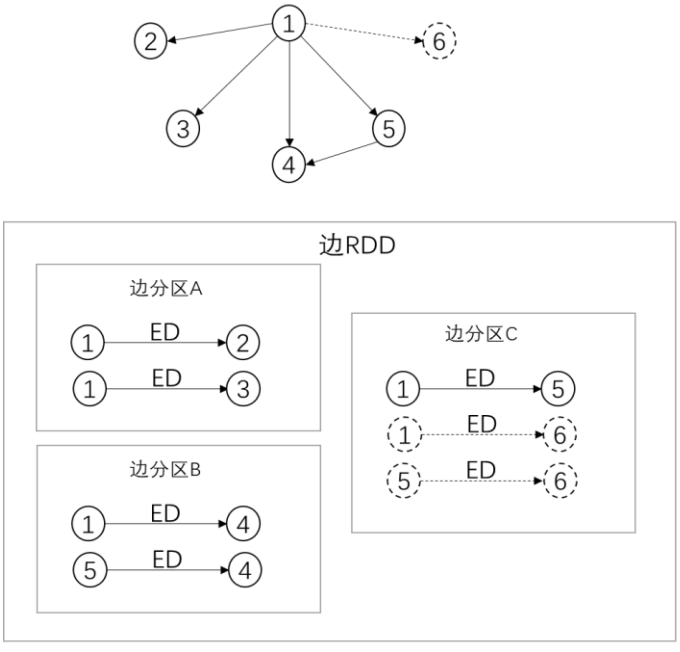
新的列簇叫做 Index，Index 并不用来存储数据，仅仅是为了将索引数据和主数据区分开来（因为在 Hbase 中同一列的数据会放在一起压缩存储）。通过设计了 version-rowkey 之间的对应关系，可以快速地找到 version 到 rowkey 的映射，将 rowkey 经过排序以后，同一个 version 开头的 rowkey 会聚集到一起，这将对我们根据 version 选择图的快照提供很大的帮助。

3.3 计算副本的研究与设计

生成的存储快照会可以转储到 Spark 的分布式内存 RDD 中，作为计算副本。RDD[44]是 Spark 独有的一个容错的，并行的数据结构，并且还支持用户显式的将数据存储到磁盘和内存中，能够灵活的控制数据分区的数量。是 Spark 进行计算的主要数据结构。

3.3.1 顶点分区和边分区的存储结构

关于图分割，主要有两个常用的分割方法。一个是点分割，另一个是边分割。Spark 采用的是点分割的方式，并且采用不同的 RDD（VertexRDD 和 EdgeRDD）存储顶点和边的集合。顶点 RDD 默认情况下会使用顶点的 ID 进行 Hash 分区，将顶点数据以多分区的形式分布在集群上。边 RDD 会按照指定的分区策略进行分区（默认情况下采用的是边的 id 进行 Hash 分区，在数据量非常大的情况下可以采用 METIS 算法进行分区，可以有效地降低通信成本），将边数据以不同的分区形式分布在集群上面。此外，为了顶点 RDD 与边 RDD 的快速通信，顶点 RDD



还存储了顶点到边 RDD 的路由表。路由表是顶点 RDD 中的一个特殊数据结构，它记录了顶点 RDD 中所有的顶点和边 RDD 的对应该关系。在计算过程中边 RDD 需要顶点的数据的时候，顶点 RDD 会根据路由表将顶点数据发送到边的 RDD 分区。顶点 RDD 和边 RDD 的分布如图 3-9 所示。

图 3-9 Spark GraphX 的顶点存储与边存储

在大部分的图计算过程中，边的计算都需要两端顶点的数据，即形成三元组的视图，例如进行 PageRank 算法的时候需要生成出边的权值，这就需要顶点将自身的权值发送给它的出边所在的 RDD 分区。Spark 会根据路由表在顶点 RDD 中生成与边 RDD 对应的重复顶点视图，根据之前的介绍，进行点切分的顶点会分布在不同的分区中。重复顶点视图的作用是作为中间 RDD，他会将顶点数据传送到相关的边 RDD 分区中。重复顶点视图的分区数量和边 RDD 相同，如图 3-8 所示，重复顶点视图 A 中存在着边 RDD 分区 A 中的所有顶点。在进行计算的过程中，Spark 会将重复顶点视图和边 RDD 进行 merge 操作，即将重复顶点视图和边 RDD 一一对应起来，从而将边数据和顶点数据组合起来，形成三元组 $\{E, V, \gamma\}$。在形成三元组的过程中，只有根据顶点 RDD 形成的重复顶点视图需要在不同的边分区之间移动，merge 操作不需要移动顶点数据和边数据。而且在大部分图中顶点数据都是远远的小于边数据的，随着迭代次数的增加，需要进行更新的边数据也越来越少，这样可以大大的减少数据的移动量，从而加快整个计算过程。

RDD 的分布如图 3-10 所示。

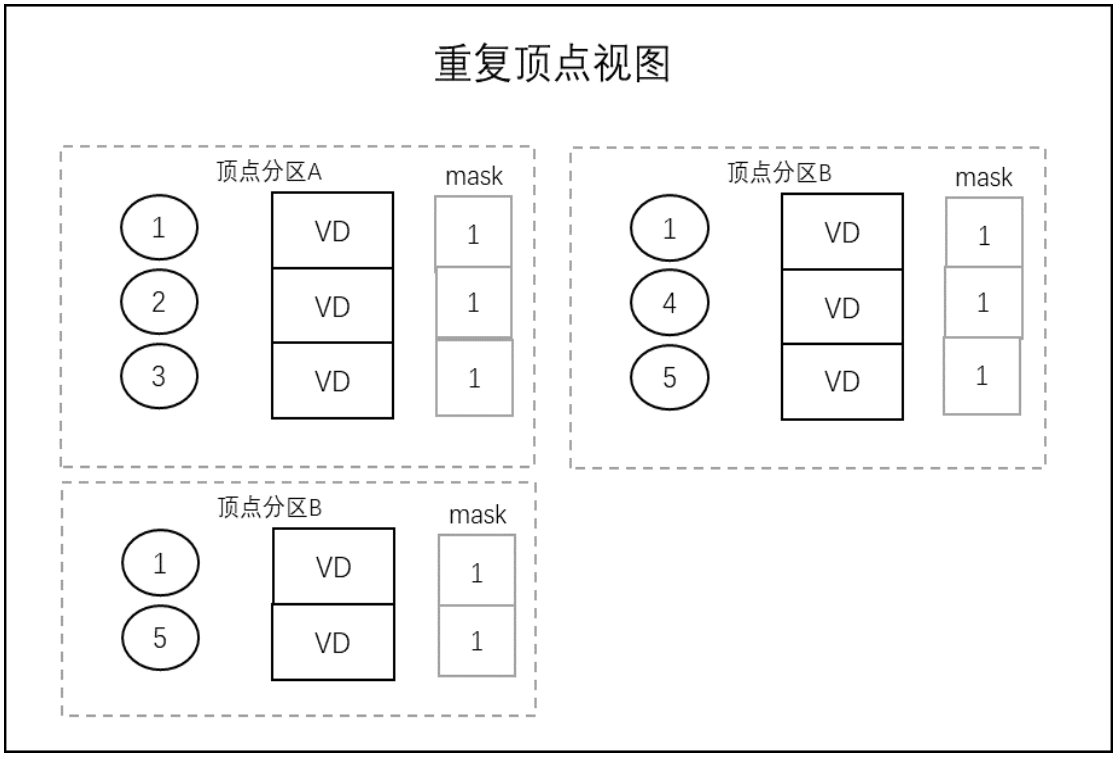


图 3-10 Spark GraphX 重复顶点视图

Spark 在顶点和边 RDD 的存储中采用数组的方式存储顶点数据和边数据，这样做可以减少访问性能的下降。Spark 还在存储图数据的时候建立了众多的索引结构，这些索引结构可以辅助快速的访问顶点数据或者是边数据。

3.4 存储主体与计算副本的同步

存储主体和计算副本分别执行了图结构的更新任务和图属性的计算任务。他们之间通过消息队列互相传递数据。从整体上说，这个过程 HBase 生成数据快照传递给 Spark，Spark 计算完以后将计算的结果返回到 Hbase 中。

存储主题和计算副本之间的同步设计图如图 3-8 所示。主要分为两个部分，第一个部分是图的导入与导出，第二个主要部分是计算请求的交互。整体交互流程如下所示：

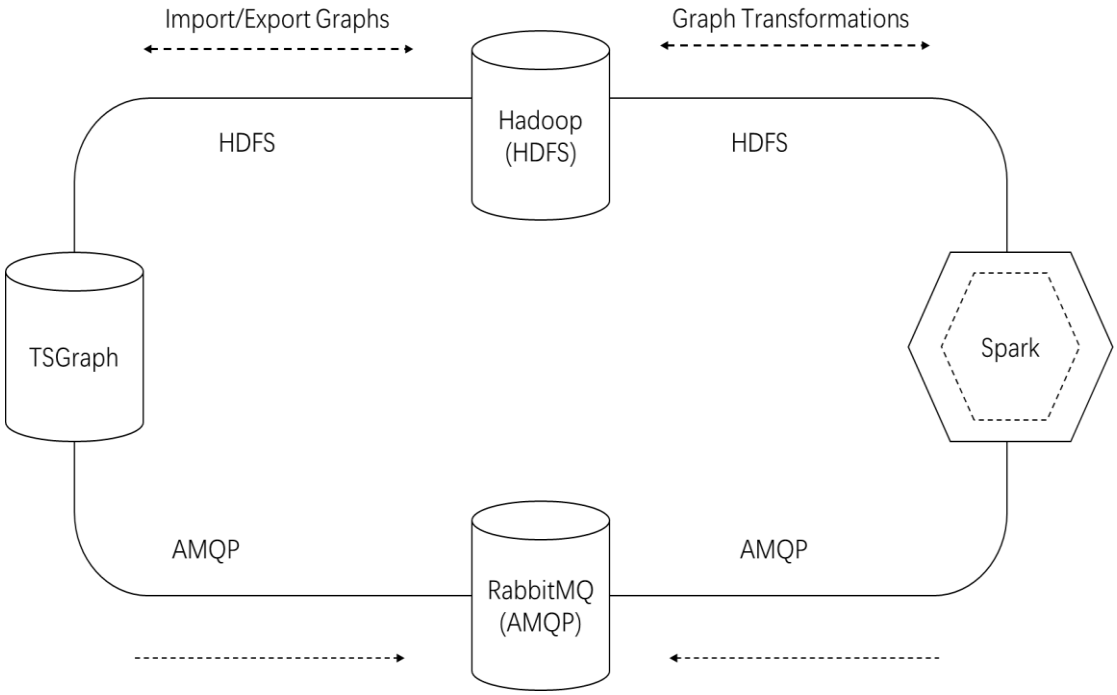


图 3-8 主题存储与计算副本的同步

- 平台发起一个计算任务，任务描述信息发送到消息队列中
- TSGraph 将相应的数据发送到 HDFS 上
- Spark 从消息队列中读取计算任务
- Spark 根据计算任务从 HDFS 中读取相应的数据，并且执行计算
- Spark 计算完成，返回相应的数据到 HDFS 中
- Spark 告知本次计算任务完成，并且继续读取下一个计算任务
- TSGraph 从 HDFS 上读取计算完的结果并且和 HBase 中的存储合并

通过以上的交互过程，有效的同步了发送请求与计算速度之间的差异，采用消息队列的方式解决了发送请求频率与计算速度不匹配的问题，极大地提升了系统的容错性。

3.5 存储体系完备性讨论

本章采用存储主体和计算副本并存的方式，有效地解决了图更新和图计算并发执行的问题，提升了整个系统的吞吐量。存储主体优秀的图存储结构使得其可以方便的进行图查询和图更新操作，计算副本采用 HDFS 和 Spark RDD 存储，极大地加快了图计算的执行速度。在主体存储和计算副本之间设置了缓冲地带，有效的解决了发起计算请求的速度和平台计算速度的不匹配，大图导入导出的速度和平台计算速度的不匹配等，增加了整个平台的可靠性。

3.6 本章小结

本章从图存储的角度设计了适合动态图计算的复杂图存储系统。首先是主体存储系统的设计，因为我们要存储的是动态时序图数据，现有的数据库都无法满足我们的需求。我们跟根据分析图查询的便利性，和时序数据存储的可行性采用了 BigTable 数据模型作为底层存储结构，同时为了图操作的便利性，我们设计实现了一个中间存储层 TSGraph，用来衔接存储后端和上层应用。然后为了查询的方便我们还在存储结构上进行了进一步的索引优化，从而方便了一个快照图的快速生成。为了提升整个系统的吞吐量，我们采用了存储主体和计算副本并存的存储方案，计算副本采用 HDFS 和 Spark 的 RDD 存储，并且使用 Spark 进行计算，不但解决了存储主题和计算副本相互冲突的问题，同时大大的提升了整个系统的计算能力。整个系统无论在功能的完备性方面还是架构的稳定性方面都有着不错的表现。

分布式的图计算系统的研究与实现

4.1 图算法的并行化结构
第四章

由第二章的介绍可以得知，BSP 是一个在图计算中广泛使用的分布式计算模型。在这里我们也采用 BSP 架构，构建图的分布式计算系统。如图 4-1 所示，计算机群中主要有两种角色，一种角色是 Master，另一种角色是 Worker。Master 作为集群的中控系统，主要负责管理集群中的 Worker，Master 会时刻保持与 Worker 之间的通信。每当一个计算任务下达的时候 Master 首先会根据指定算法进行图分区，然后将分区后的子图分别分配给参加计算的 Worker。每个 Worker 都会向 Master 注册自身的信息，Master 中维护着 Worker 详细信息的状态表，Mater 和 Worker 之间保持着固定频率的心跳，每当 Worker 的状态发生改变以后就会通知 Master，Master 会相应的修改 Worker 详细信息表中该 Worker 的信息。

在每个超步的计算过程中，Worker 都会首先遍历自身的内存，计算内存中每个节点的值。每个 Worker 计算完成后会暂时休眠，等待其他 Worker 也完成自身任务的计算，当所有的 Worker 都计算完毕以后，他们之间会进行互相通信，计算出关联节点的信息，最后统一由 Master 更新所有节点的值。

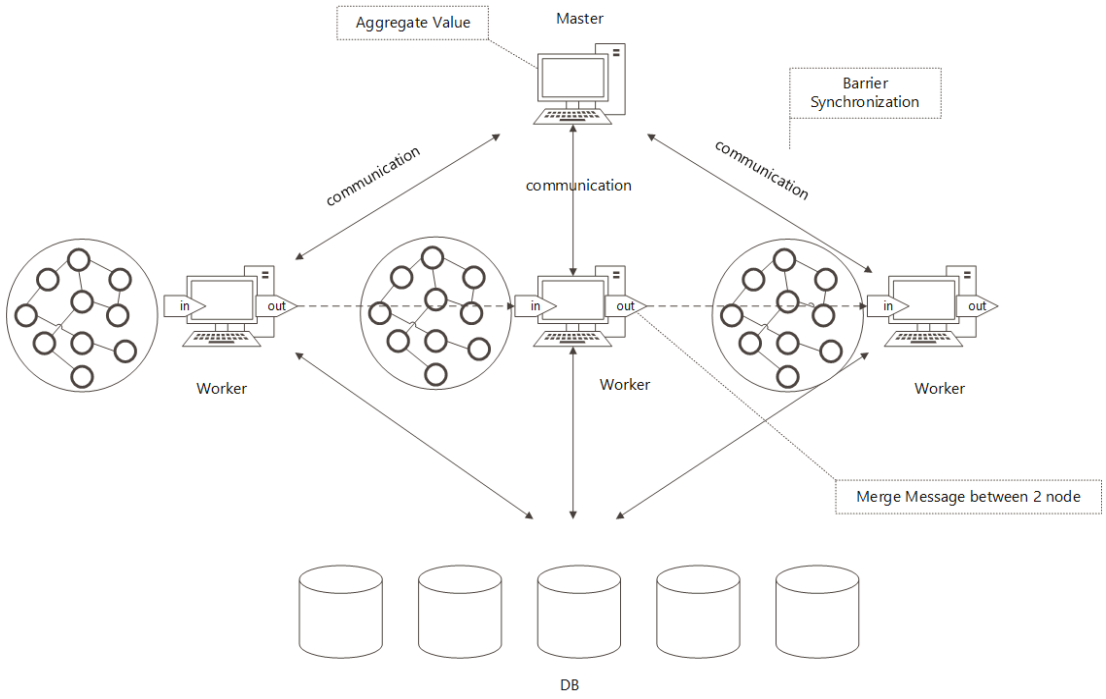


图 4-1 图计算分布式并行框架

采用该计算框架的意义是图分割后的数据可以存储在不同的 Worker 上，在良好的图分割模型下可以有效地减少 Worker 之间的通信量。图中的顶点之间的

计算采用纯消息传递的方式，每个 Worker 之间异步计算，Worker 之间采用批量消息传输的方式交互信息，整个系统的执行效率大大提升。

以 PageRank[45]算法来举例说明并行图算法的具体执行过程。PageRank 最初只是用来计算网页重要性的，后来逐渐演化成了衡量网络中节点重要性的一个重要指标。

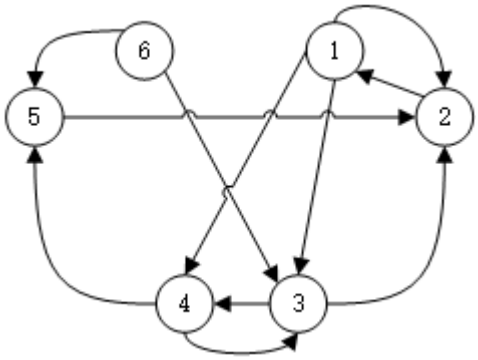


图 4-2 PageRank 示例网络

如图 4-2 所示，可以认为是一个现实网络中大图的简化。图中有 6 个节点和 11 个关系。通过执行 PageRank 算法，会得到一个各个顶点的概率分布，用来表达链入任意一个节点的可能性。PageRank 的计算过程开始时图中的各个节点概率分布是相同的，需要经过多轮迭计算他们的真实值概率分布，通过多轮迭代可以不断地调整每个顶点近似的 PageRank 值，从而使计算结果更接近于各个节点的理论真实值。

下面我们看一下 PageRank 算法在 BSP 并行框架上的具体运行过程。系统接收到一个计算任务之后首先会将任务分配给 Master，再由 Master 统一调度。Master 会先根据指定的图分割算法将大图分配到各个 Worker 上，同时向 Worker 下达计算任务。

针对图来说，系统接收到该图之后首先会将该图划分为 3 个部分。节点 1，2 被划分到 Worker1，节点 3，4 被划分到 Worker2，节点 5，6 被划分到 Worker3。每个 Worker 中维护着本地节点和其他 Worker 中有关联节点的关系。

- Compute，计算本地节点数据的过程，包括合本地其他节点传递过来的消息和分配要传递给出边节点的消息。Worker1 节点会计算本地内存中的 1，2 节点值，Worker2 会计算本地内存中的 3，4 节点的内容，Worker3 相应的计算本地内存中 5，6 节点的内容。
- Send，向当前节点的出边节点发送消息。如图所示，Worker1 中的节点 1 会向节点 2，3，4 发送消息，节点 2 向节点 1 发送消息。Worker2 中的节点 3 分别向节点 2，4 发送消息，节点 4 分别向节点 3,5 发送消息。Worker3 中的节点 6 向节点 3，5 发送消息，节,5 向节点 2 发送消息。

- Barrier，同步边界。在系统中不同的 Worker 异步的执行自身的任务，当一个 Worker 执行完自身任务时会先陷入休眠状态，等待其他的 Worker 执行自身的计算任务。当所有的 Worker 都执行完自身的计算任务之后，不同的 Worker 之间会互相通信，同步计算结果到关联的的 Worker 中。
- Parse，节点上消息的聚合。每个 Worker 上的节点进行消息的聚合。Worker1 中节点 2 接收来自节点 1,3,5 的消息。节点 1 接收来自节点 2 的消息。Worker2 中节点 3 接收来自节点 1,4,6 的消息，节点 4 接收来自节点 1,3 的消息。Worker3 中节点 5 接收来自节点节点 4,6 的消息。

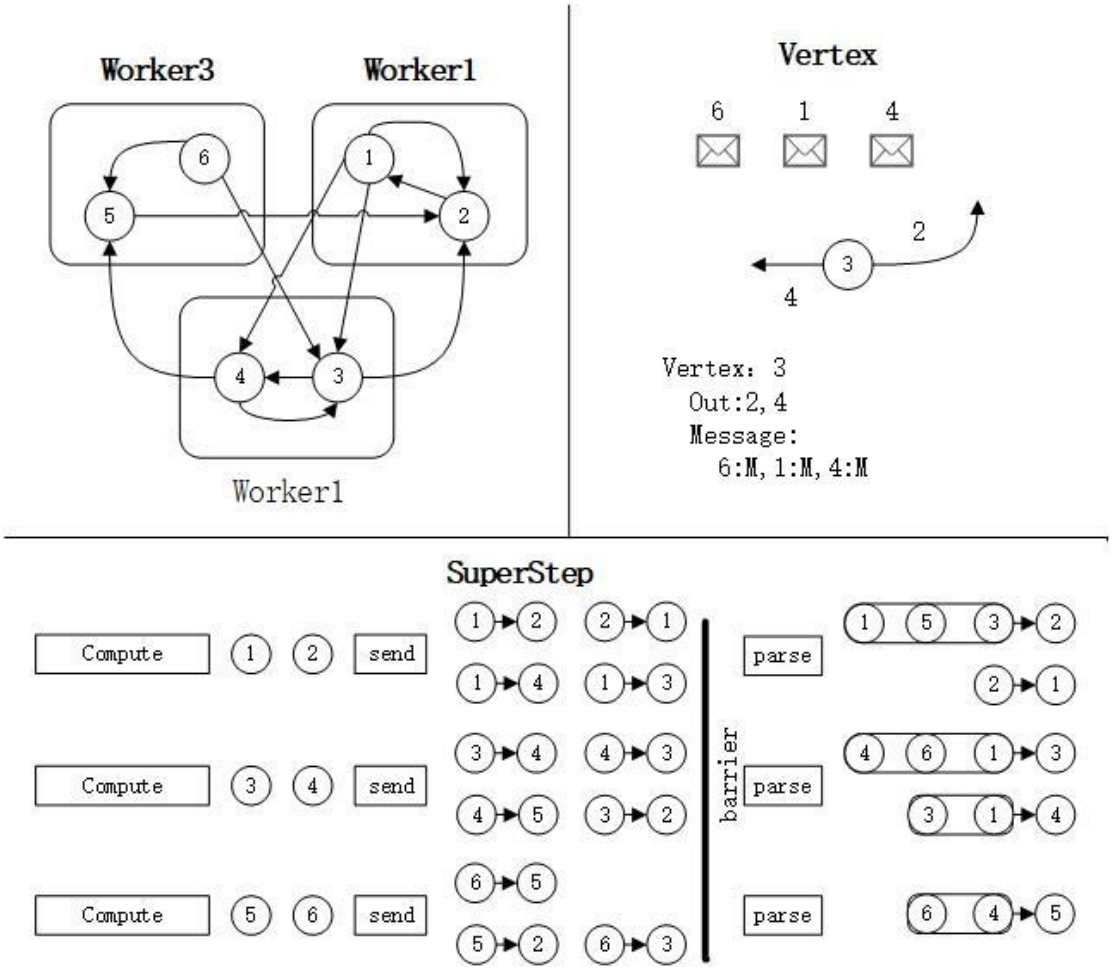


图 4-3 基于边分割的 BSP 模型 PageRank 计算

如图 4-3 所示，左上角的部分是该系统中不同 Worker 上的内存模型。右上角的部分是计算过程中一个单独的顶点需要执行的行为。下面是该计算过程中每个超步的具体执行过程。该模型虽然可以完美的实现图的分布式计算，但是潜在的问题也是非常大的，每个 Worker 之间的信息同步太多，严重的影响到了系统的整体执行效率。

4.2 图算法并行化结构的优化

为了进一步优化该并行化结构，可以着重修改 Worker 之间的信息同步花费上。图 4-2 采用的是边分割（Edge Cut）的图分割方式，采用边分割的好处是可以减少存储的花销。但是带来的影响就是通信花销会变大，在这个存储资源代价远低于计算和网络代价的时代，这个优势可以忽略不计。因此可以考虑另一种图的分割方式，点分割（Vertex Cut），点分割的方式是一个节点可以跨多个机器存储，这些个存储里有一个主存储（MainStore）和若干个副本存储(Replica)。每个 superstep 计算完之后我们只用同步跨机器存储的 MainStore 和 Replica 即可，可以有效的降低通信代价。具体操作如图 4-4 所示：

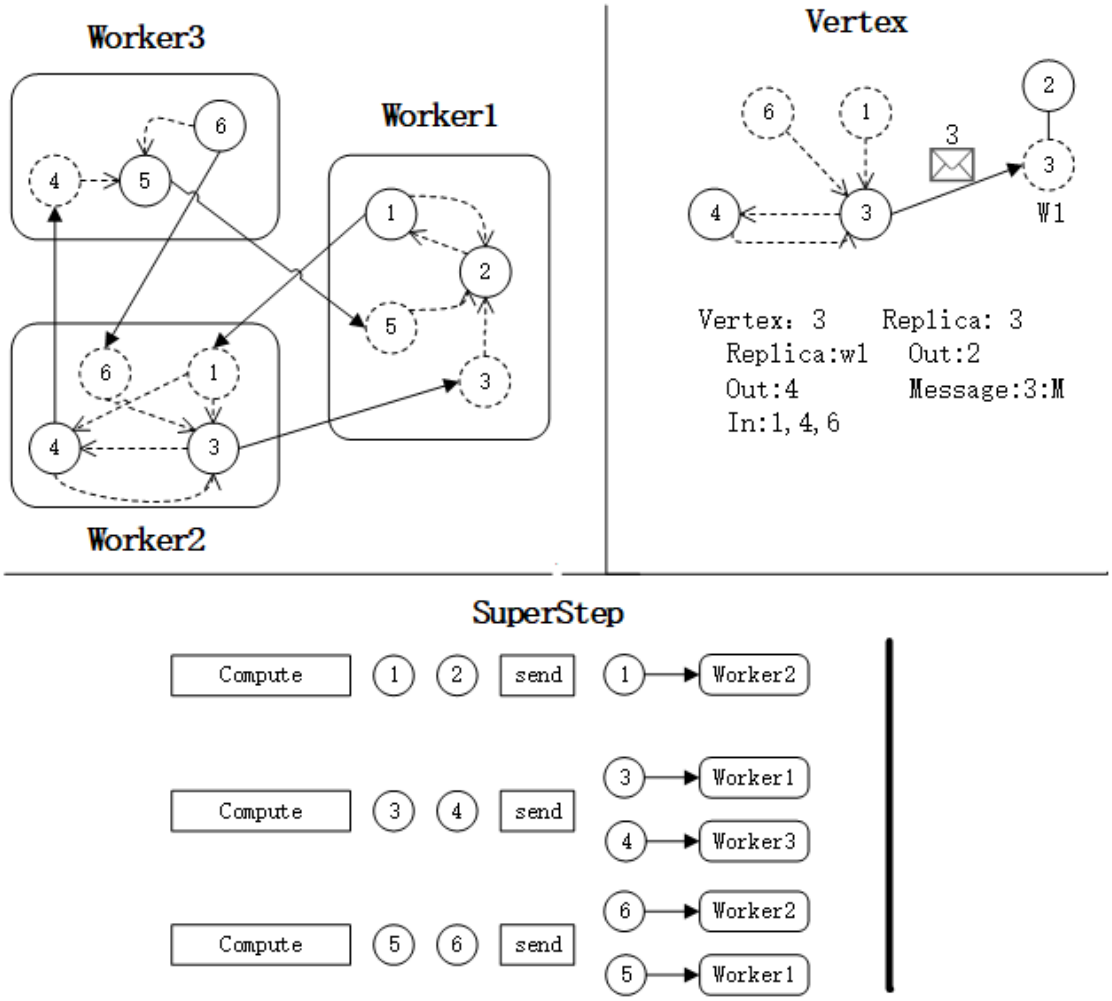


图 4-4 优化后的 BSP 模型 PageRank 计算

改进的 BSP 图计算模型执行过程如下，系统接到计算任务之后会把数据分成三份，并且被切分的节点要生成一个备份。如图 4-4 所示，Worker1 存储节点 1，2 并且存储节点 3，5 的 Replica。Worker2 存储节点 3，4 的 MainStore，并且存储节点 1，6 的 Replica。Worker3 存储节点 5，6 的 MainStore 并且存储节点 4 的 Replica。

- Compute，计算本地节点数据的过程，包括本地其他节点传递过来的消息和分配要传递给出边节点的消息。Worker1 中节点 1 接收来自节点 2 的消息，并且发送消息给节点 2。Worker2 中节点 3 发送消息给节点 4，并且接收来自节点 1,4,6 的消息。节点 4 接收来自 1,3 的消息，并且发消息给节点 3。Worker3 中节点 5 接收来自节点 4 的消息，节点 6 发消息给节点 5
- Send，被分割的节点由 MainStore 向 Replica 发送消息。
- Barrier，不同的 Worker 同步消息，主要是被分割的节点 MainStore 和 Replica 进行消息的同步。

通过上面的对比可以看出，使用点分隔的方式可以大大地减少不同 Worker 之间的通信代价，使更多的计算都在本地完成，极大地提高了系统的执行效率。

4.3 并行化增量式算法的研究与实现

动态图计算是为了适应快速变化的网络结构的一种实时计算方式，动态图计算的原理是，每进行一次新的计算都利用到原有的计算结果，从而减少总体计算规模，可以更加快速的生成最新的计算结果。动态图计算将动态图存储为阶段性的静态图组合，$\mathrm{G} = \{G_{t1}, G_{t2} \ldots G_{tn}\}$表示该图在t1到tn的动态集合。$\mathrm{t1}, \mathrm{t2} \ldots \mathrm{tn}$如上一节介绍，是一系列的非重合时间区间。

动态图的计算过程是：

步骤 1：计算两个时刻静态图的区别。标记$G_{t1} - G_{t2}$为旧的节点，记为G_{old}，标记$G_{t2} - G_{t1}$为新发生改变的节点，记为G_{new}。

步骤 2：首先设置G_{new}为活跃顶点，设置G_{old}为非活跃顶点。

步骤 3：从每个活跃顶点开始计算，并且将计算结果传递到相邻顶点。

步骤 4：对每个顶点检查旧值与新值之间的diff，当$\mathrm{diff} > \mathrm{detla}$则设置该节点为活跃节点，detla 为阈值，否则设置该节点为非活跃节点。

步骤 5：重复步骤 3 到 5，直到所有节点都变为非活跃定点。

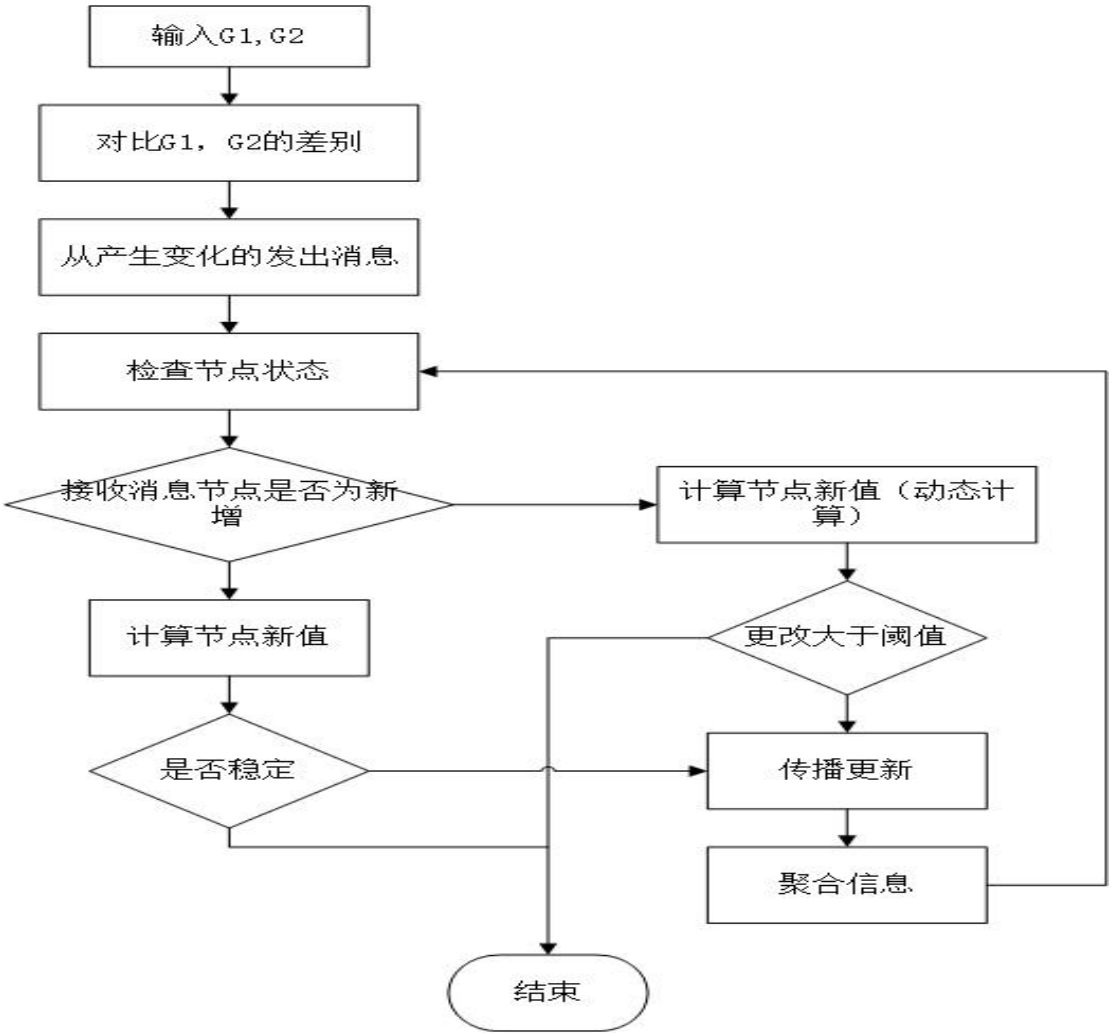


图 4-5 动态计算流程图

动态图算法相对于全量静态图算法最大的特点是，对于动态图的计算，每次只要计算上一个时间和当前时间中整个图改变的部分，从而在计算一开始的时候可以只激活变化的那部分节点，而全量静态图算法则要激活所有的节点。因此动态图算法相对于全量静态图算法在图规模非常大，并且每次变化的节点数量很少的情况下具有很大的优势。动态图算法只对两个时间产生变化的那部分数据敏感，计算时间不会随着整个图的规模增大而线性增加，这对于计算资源有限的情况是非常有益的。

在以上背景下，本文研究并实现了一个基于分布式图处理框架的动态图算法，嵌入的图算法主要包括 PageRank，TrustRank[46]和 ShortestPath 等。分布式动态图算法的两个设计难点是，消息传递方式和同步模型的设计，在本文中我们采用触发式消息传递模式和大整体同步模型，每个计算节点内进行异步计算，不同的计算节点之间再进行整体同步。

动态图的计算是根据触发机制执行的，大图中发生变化的节点或者边会直接影响到其周围的节点或者边。用计算 PageRank 算法来举例，图 4-5 是一个 PageRank 算法动态计算的消息传播过程。PageRank 最初是用来计算网页重要性的，后来逐渐发展成为衡量网络中节点重要性的一个重要指标。

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

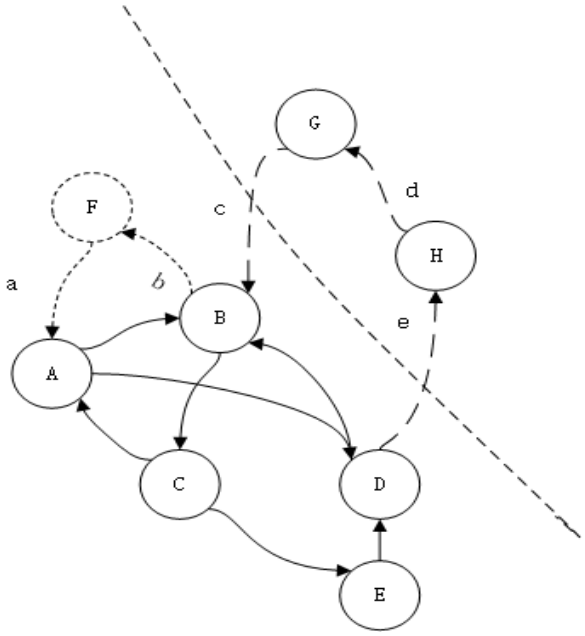


图 4-5　PageRank 动态计算消息传递

PageRank 的计算过程中每个节点都会出边指向的邻居节点传递自己的重要性贡献，直到整个网络中的每个节点重要性趋于稳定为止。在G_1时刻整个图的节点包括$\{A, B, C, D, E, F\}$，G_2时刻新增了节点$\{G, H\}$，新增了边$\{c, d, e\}$，而且减少了节点$\{F\}$，减少了边$\{a, b\}$。对应的 PageRank 的动态算法执行过程为，首先$\{G, H, F, a, b, c, d, e\}$作为发生改变的节点和边，会直接影响到与他们直接相连的节点$\{A, B, D\}$。节点 F 以及边$\{a, b\}$的消失主要影响到节点$\{A, B\}$，对于节点 A 来说，A 需要减去来自节点 F 的重要性贡献，然后 A 用新的重要性值和旧的重要性值比较，如果阈值大于 detla 则进一步将影响力传递到$\{B, D\}$。对于节点 B 的影响则是 B 将要重新划分影响力的传递，将原先传递到 F 的影响力也传递到 C。节点$\{G, F\}$的增加和边$\{c, d, e\}$的增加则主要影响到节点$\{B, D\}$，对于节点 B 来说，主要是接收从节点 G 经由边 c 传递过来的影响力，变化值高于 detala 则要进一步传播。对于节点 D 来说，需要将自身的影响力传递给 H。

对于 PageRank 这种迭代式算法，利用动态计算方式可以在计算初始只激活与更改部分相关的节点，利用差量的计算方式，以及一个变化阈值 detla，可以

在一个节点的差量变化小于 detla 时停止进一步传播，从而加速整个计算过程。算法的伪代码如下所示。

算法：PageRank 动态图算法

输入： 目标图 G，其时间区间划分为$\{G_{t1}, G_{t2} \ldots G_{tn}\}$

输出： 一系列时刻计算结果的集合

SendMessage

1: input(G_t, G_{t+1})
2: $G_{old} = G_t \cap G_{t+1}$
3: $G_{new} = G_{t+1} - (G_t \cap G_{t+1})$
4: $G_{active} =$ active(G_{new})
5: **for** 任意的 $v_j \in G_{active}$ **do**
6: **if** abs$\left(v_j^{prev} - v_j^{cur}\right) > detla$
7: **for** 任意的 v $\in$ v_j*的邻居*
8: **if** v $\in$ G_{old}
9: SendMessage($\frac{\left(v_j^{prev} - v_j^{cur}\right)}{degree}$)
10: **else**
11: SendMessage($\frac{v_j^{cur}}{degree}$)
12: **end if**
13: **end for**
14: **end if**

ReceiveMessage

1: **if** v $\in$ G_{old}
2: **if** message > detla
3: update(v); active(v)
4: **else**
5: update(v)
6: **end if**
7: **else**
8: **if** message $- v_{cur} > detla$
9: update(v); active(v)
10: **else**
11: update(v)
12: **end if**
13: **end if**

Dynamic Algorithm

1: while $G_{active}.count > threshold$
2: SendMessage
3: ReceiveMessage
4: end while
5: output(G_{t+1}^{new})

另一个图计算中常用的算法即最短路径，在地图中的路径规划，社交关系网络中的好友推荐，最短路径是一个常用的图算法。在这里我们采用一种在大规模网络中比较实用的一种基于地标的最短路算法[47]。该算法的特点是采用一些节点S作为地标集合。图中的每个节点都会维护与地标集S的最短距离。这样求图中的任意两个节点v1和v2的最短距离就会演化成求v1到地标集S的最短距离加上地标集S到v2的最短距离。该算法已经证明在地标选取合理的情况下是有效的。

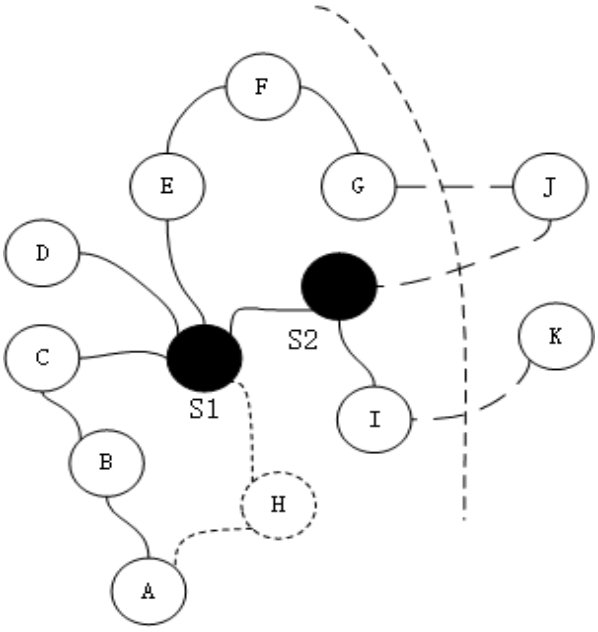


图 4-6　Shortest Path 动态计算消息传递

如图4-6展示了基于地标的最短路算法动态计算过程。$\{S1, S2\}$是我们选取的地标集。在G_1时刻，图中的顶点主要包括$\{A, B, C, D, E, F, G, H, I\}$。在$G_2$时刻图中新增了点$\{J, K\}$，以及减少了点H。对于点H的消失来说，首先我们会检查点H的邻居节点，查看H的邻居节点到地标集S的最短距离是否有更新。可以得到A到S1的最短距离从2更新为3，A到S2的最短距离从3更新到4。然后继续检查A的邻居节点是否需要更新，如果不需要更新则停止传播。对于新增的节点J来说，检查J的邻居节点G到地标集的最短距离是否需要更新。G到S2的最短距离由4更新到2，G到S1的最短距离不需要更新。再检查G的邻居节点，以此类推。算法的伪代码如下图所示：

算法：ShortestPath 动态图算法

输入：目标图 G，其时间区间划分为$\{G_{t1}, G_{t2} \dots G_{tn}\}$，地标集$S$，$\{S_1, S_2 \dots S_n\}$

输出：一系列时刻计算结果的集合

SendMessage

1: input(G_t, G_{t+1})

2: $G_{old} = G_t \cap G_{t+1}$
3: $G_{new} = G_{t+1} - (G_t \cap G_{t+1})$
4: $G_{active} =$ active(G_{new})
5: **for** 任意的 $v_j \in G_{active}$ **do**
7: **for** 任意的 v $\in$ v_j*的邻居*
8: **if** v 到 S 的距离更新
9: SendMessage(v*到S的距离* + 1)
12: **end if**
13: **end for**
14: **end for**

ReceiveMessage

1: **for** message $\in$ Messges
2: **if** message < v 到 S 的最短距离
3: update(v); active(v)
13: **end for**

Dynamic Algorithm

1: while $G_{active}.count > threshold$
2: SendMessage
3: ReceiveMessage
4: end while
5: output(G_{t+1}^{new})

动态图的计算相对于静态图计算的的最大特点就是触发计算的方式，动态图采用更新，触发的方式驱动整个计算的进行，因此我们设计出三个重要的函数。首先是 SendMessage 函数，SendMessage 是整个计算的发起点。在每次动态计算中两个不同时刻的图快照会先进行快速对比，对比得到两个图之间的差异之后就会将改变的节点设置为活跃，然后由活跃节点向邻居节点发出信息。另一个重要的是 ReceiveMessage 函数的设计，ReceiveMessage 函数主要负责从邻居节点接收信息，将接收到的信息和节点本身的信息合并，并且更新节点本身信息。AggreateMessage 函数主要负责将不同计算节点上得到的结果聚合。

整个算法的运行过程是，首先在每个 Worker 上检查图中的活跃节点，由活跃节点开始 SendMessage。然后活跃节点影响到的节点会 ReceiveMessage，并且和自身的原有信息合并。最后不同的 Worker 之间会相互通信，做全图聚合处理。

4.4 增量式算法效果评估

阿斯顿我们在 Spark GraphX 图处理框架上实现了上述的分布式动态图算法。整个系统部署在一个四个节点的集群上运行，每个计算节点的配置为 64 GB

DDR3 内存和 3.10 GHz Intel Xeon E3-1220 v2 CPU，操作系统为 Ubuntu16.04，我们使用了从斯坦福大学公开的 SNAP[48]上获取的 4 个开源数据集。数据集的详细信息如表 1 所示，主要包括数据集名称、图类型、节点数目和边数目。

表 1： 实验所用数据集

数据集	类型	节点数目	边数目
wiki-Vote	有向图	7715	1036892
soc-Slashdot0811	有向图	77360	905468
email-EuAll	有向图	265,009	420,045
web-Google	有向图	875,713	5,105,039

首先本文在表 1 数据集上对 PageRank 的动态图算法上进行了实验。数据规模从 7000 多到百万级别，分别以 12%的增量规模进行多次验证。如图 5 所示，图中横轴代表的是超步次数，纵轴代表的是每个超步被激活的节点数量。

从图中可以看出，使用动态算法计算的方式，每次初始阶段只用激活少量的节点，这种方式可以大大的加快收敛速度。从具体效果来看，除了 wiki-Vote 数据集上动态图算法的性能提升不明显，其他几个数据集上都使整体收敛速度提升了 3 倍左右。证明了使用动态算法的确可以极大地加速算法的收敛，证明了该算法在提升算法效率方面的有效性。

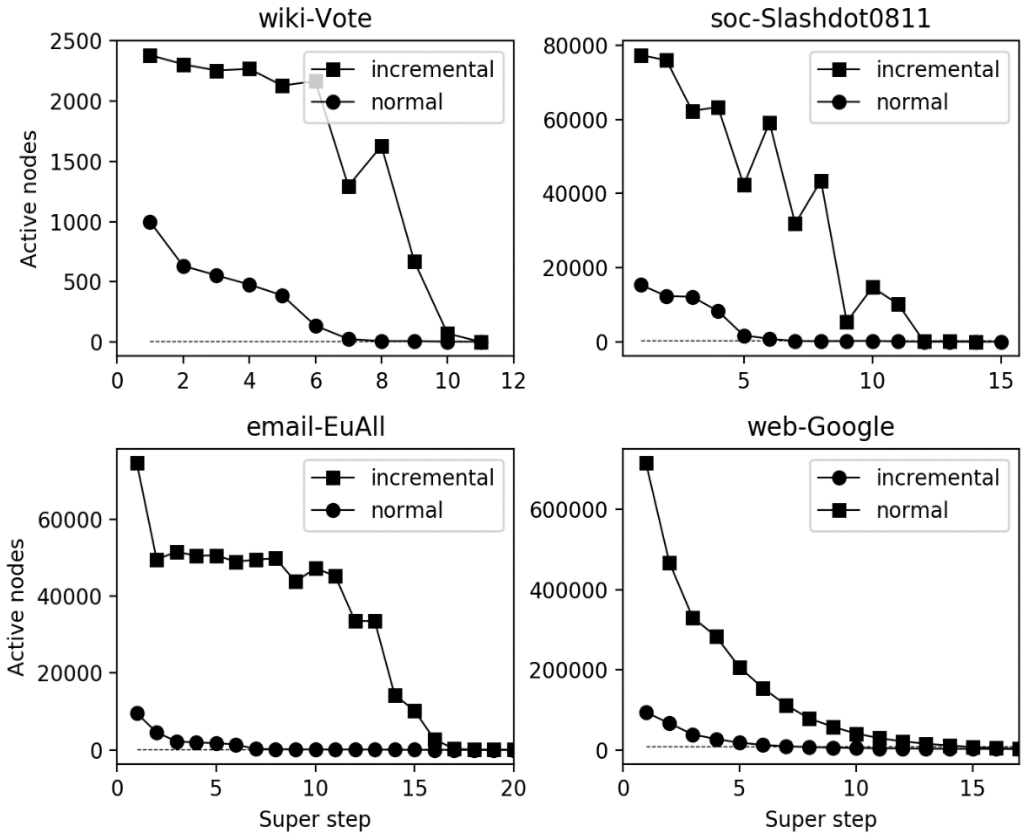


图 4-7 在不同数据集下 PageRank 动态式算法和普通算法的性能差异

另一方面，动态算法使用了触发、传播的计算方式。而触发的方式是根据值的改变是否大于阈值而决定的，这就会造成一定的准确度丢失。下面一个实验，我们在 web-Google 数据集上分别用不同规模的增长量验证了该算法在准确度方面的表现。

如图 4-8 所示，我们采用 500K 的节点作为已有原始图，使用其他 300K 的节点作为增量图。分别采用了 2%、4%、8%、16%、32%的增量方式进行了多次实验。实验结果如图中所示，随着增量规模的增加，准确度的变化呈现了一个先增后减的态势。在节点规模增加到 800K 的时候，8%增量的准确度最高，其次依次是 4%增量、16%增量和 2%增量，32%增量的表现最差。为我们在工程实践中的具体使用提供了依据。

实验体结果表明，动态图计算相较于全量静态图计算在计算效率上有很大的优势，但是同时也会带来一些准确度的损失。同时也表明，随着时间的推移，我们需要使用全量式算法对动态计算的结果做校正，从而保证结果不至于产生太大偏差。

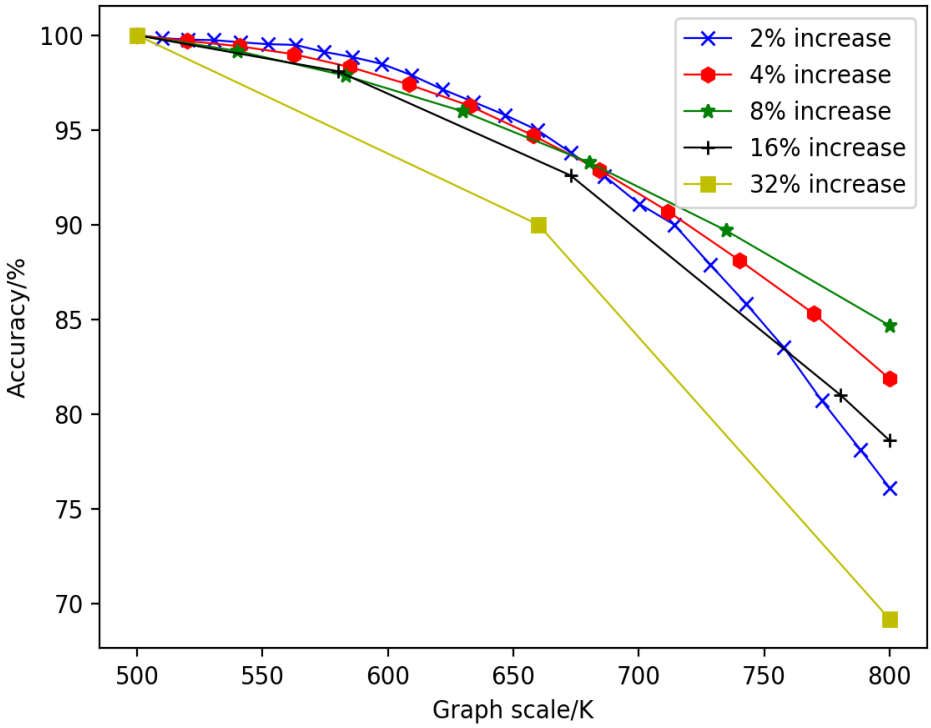


图 4-8 PageRank 算法的准确度与增量数据的规模关系

然后本文在以上数据集上分析了 ShortestPath 动态图算法的性能，ShortestPath 不需要迭代式计算，这就注定了 ShortestPath 的计算模型与 PagrRank 不一样，ShortestPath 动态图算法的传播过程不存在衰减的情况，从而不会造成准确度方面的损失。

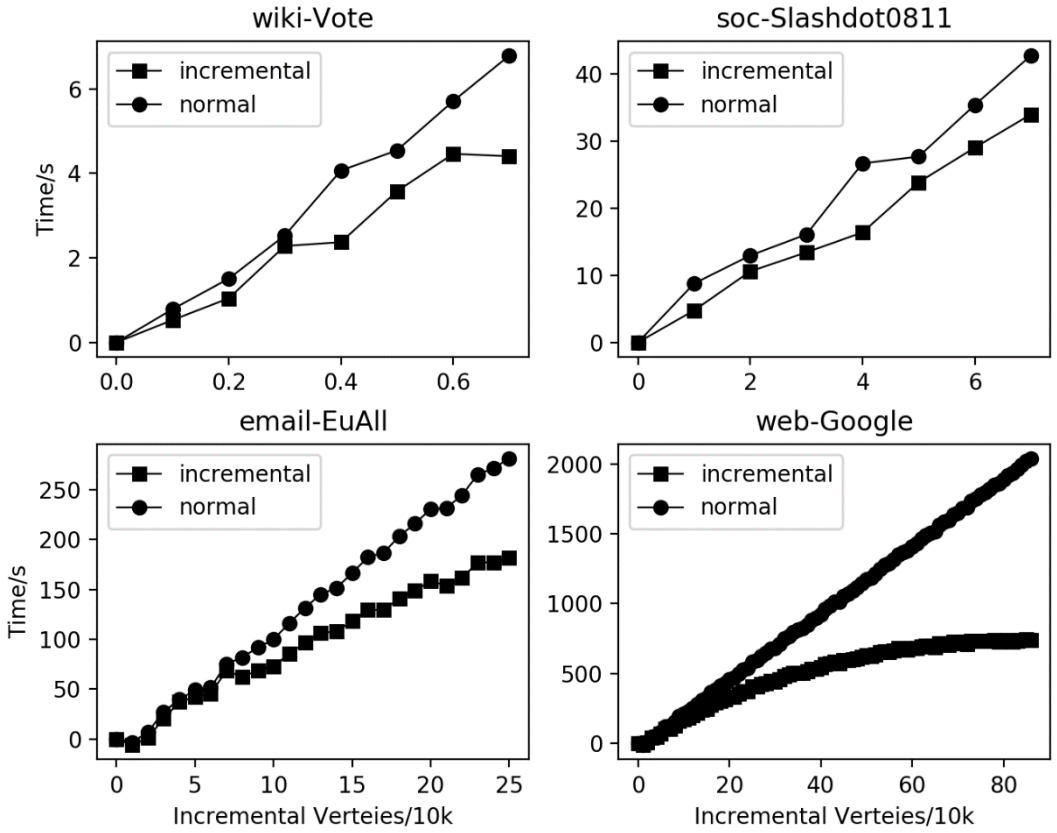


图 4-9　在不同数据集下 ShortPath 动态式算法和普通算法的性能差异

图 4-9 分别是 ShortestPath 动态图算法在是个数据集上进行动态式计算和普通计算的时间效率对比。图一每次计算的增量数据为 1k 节点，图二、三、四的每次增量均为 10k 节点。从图中可以看出在计算的初始阶段，两种算法的时间几乎没有差别，但是随着已有数据的增加，动态式算法的效率明显高于普通算法。

从图一、二、三看的话，两个算法的效率貌似都是线性增加的。但是从图四看来动态式算法的效率曲线更接近于对数分布。原因是前三幅图的增量数据较少，从图上看来更接近于线性分布。从长期的增量计算来看，动态式算法的效率曲线确实更接近于对数分布。说明了随着增量数据的增加，动态式算法的效率要明显的好于普通算法。

4.5 小结

本节基于 Spark GraphX 实现了一种基于动态图计算的算法，并且在多个大规模数据集上进行了验证，数据规模从几千到上百万。根据实验结果来看，我们

的算法对于大图计算的时间效率带来了较大的提升，特别是 PageRank 算法，效果提升尤为明显。对于 PageRank 来说虽然同时也有准确度方面的损失，但是只要定期使用全量式算法进行更正，还是可以满足很多实时系统的需求的。同时针对准确度损失这方面，从实验的结果来看，过大或者过小的动态更新都会造成准确度的下降。在这方面，未来希望可以在本文的基础上作进一步研究，选择适合的全量更新点，让 PageRank 动态图算法可以获得更好的效果。

分布式图计算分析平台的设计与实现

5.1 总体设计
第五章

本章首先介绍了该分布式图计算平台的详细架构。然后详细描述了各个模块的详细设计过程，最后展示了系统的用户操作界面。

5.1.1 系统架构

根据前几节的介绍，我们实现了一个完整的分布式图计算平台，在这里我们叫做 OpenGraph。OpenGraph 封装了前几节我们提到的各种功能，包括时序图的存储能力，动态图的实时计算能力，图关系的快速查询能力，还有基于历史的分析能力，下面我们简要介绍一下 OpenGraph 的主要架构。

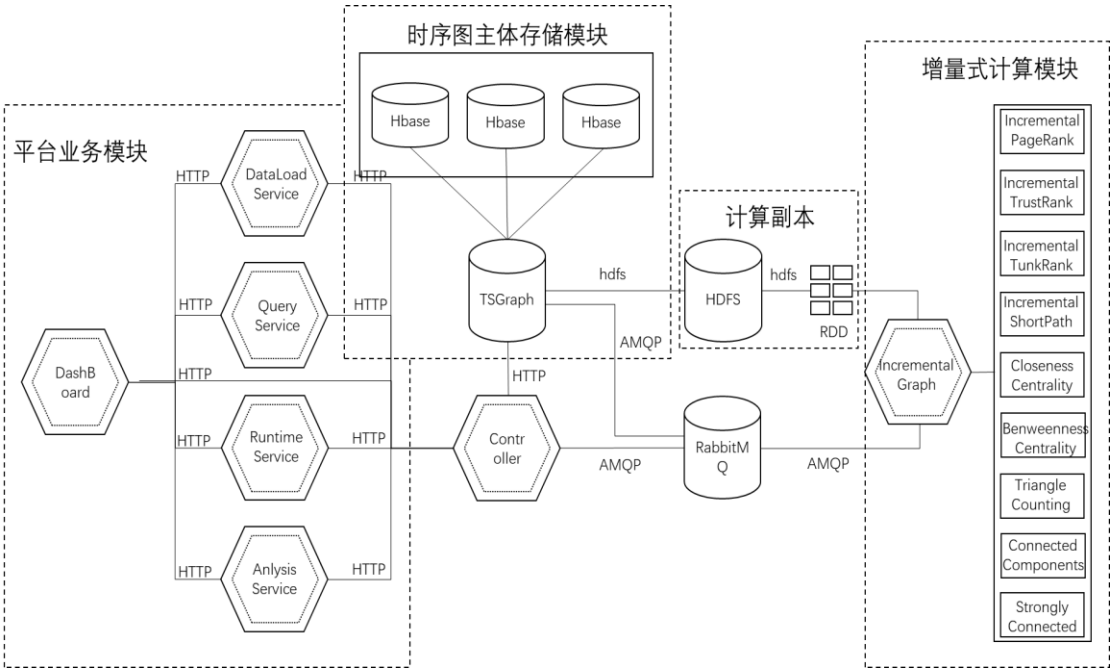


图 5-1 动态计算平台架构图

OpenGraph 主要有 4 个组成模块，分别是底层的时序图存储模块，时序图存储模块是基于 HBase 实现的一个时序图存储模型，主要由 HBase 做后端存储，TSGraph 上层封装，主要负责执行常见的查询任务，特点是响应速度快，准确率高。第二个模块是计算副本模块，计算副本模块是为了实现存储与计算隔离而专门设计的模块，该模块主要由 Hadoop HDFS 和 Spark RDD 构成，主体存储模块的数据会定期以快照的形式发送到 HDFS 上，Spark 计算时会读取 HDFS 上的数据并且按通信代价最小的目标做分割，以 Spark RDD 的形式存储在 Spark 集群的内存上，供 Spark 调度计算。第三个模块是增量式计算模块，在该模块中我们实

现了增量式算法，因为在图的实时分析过程中随着图的规模增大全量式算法会非常浪费时间，增量式算法可以在计算初始阶段只用激活少量的节点从而加快整个计算过程，在可容忍的准确性损失范围内，增量式算法可以较大的提升性能。最后一个模块就是我们的上层封装应用，我们基于以上三个模块搭建了一个易用的图计算平台，该计算平台主要支持图的增量式导入，图的增量式计算，基于图的查询还有在图上基于历史事件的分析。

5.2 详细设计与实现

5.2.1 主体存储模块

主体存储模块主要分两个部分，后端 HBase 和上层 TSGraph。后端 HBase 用来负责图数据在磁盘上的存储。为了快速的检索图数据，我们设计了适合存储图数据的 BigTable 存储模型，通过该存储模型和 Hbase 中 RowKey 的优化，使得单个节点检索及一层关系检索降低到了毫秒级别，完全可以满足日常中的大部分检索任务。对于深层次的图检索，我们在 Edge 的存储上设置索引，并且采用广度优先搜索和深度优先搜索结合的方式，在百万级别的图上可以快速地检索到 10 跳以内的图关系。

基于 HBase 的后端存储我们实现了一个 TSGraph（Time Series Graph）的上层封装，TSGraph 的作用是让我们用图的思想来进行图上的查询。该查询方式借鉴了 Gremlin 语言，该语言是一种 DSL 语言。TSGraph 还实现了时序图的检索，通过在 HBase 上将时间属性和 Vertex 联合作为 RowKey 的设计，时间序列相同或者近似的数据会存放在物理位置靠近的位置，查询的时候通过顺序加载可以方便的加载出一个时刻的静态图，使得静态图的查询速度大大加快。TSGraph 是和平台业务交互的一个主要模块，也算是一个中间模块。有了 TSGraph 的存在，上层平台业务不需要知道底层到底有什么东西，只用向 TSGraph 提交计算任务即可。

同时 TSgraph 还负责和计算副本之间的交互，每当一个计算任务产生的时候，TSGraph 会向计算副本提交一个所需存储的快照。当计算副本得到计算结果的时候会重新返回到 TSGraph 中，TSGraph 此时还要负责将返回的数据和主题存储中的数据做 Merge（因为计算过程中主题存储的数据也可能发生变化）。Merge 成功以后会修改主题存储的数据。

5.2.2 计算副本模块

计算副本模块主要由 HDFS 和 Spark RDD 两个模块。HDFS 主要用来存储一些中间计算结果，Spark RDD 则是主要用来工计算任务的调用。计算副本模块主要连接两个模块，一个是主体存储模块，另一个是增量计算模块。

计算副本模块主要是为了与主体存储模块的数据隔离开来，一方面是这样的设计可以增大整个计算的吞吐量，同时也减少了实时计算任务的延时。计算副本模块和主体存储模块与增量计算模块的交互是这样的，每次 TSGraph 收到计算任务，会先根据任务的需要加载需要计算的数据。加载完需要的数据之后会存储到 HDFS 上，如果该任务是一个新的计算任务则 HDFS 直接存储。如果该任务是一个增量式计算任务则要将本次加载的数据与 HDFS 上本身的数据先进行 Merge 操作，Merge 操作的主要作用是为了寻找两次数据之间的差异，从而确定本次增量式计算一开始要激活的节点。进行该步骤操作之后，整个图中的数据会被标记为两部分，一部分是未发生变化的节点，记做 Gold，另一部分是发生改变的节点，记做 Gnew，增量式计算首先会从 Gnew 开始触发计算。

当 Spark 要进行计算时会先将 HDFS 上处理好的数据加载到 Spark RDD 上面。Spark RDD 会根据上一步已经做好的标记确定计算的执行过程，在整个迭代式计算的过程中，如果内存一一直可用，则整个计算过程中的中间计算结果都会一直存放在 Spark RDD 上。如果计算过程中内存不够，此时还可以将部分计算结果存放在 HDFS 上，下次计算的时候再从 HDFS 上读取。

采用存储与计算隔离的方式，一方面是增加了整个系统的吞吐量，另一方面也降低了一个存储的负载，减少了读写锁的操作，大大地简化了系统的复杂性。

5.2.3 增量计算模块

增量计算模块是整个动态图计算的核心，采用增量式计算的思想，使得大规模图的实时计算变为了可能。更好的反映了现实世界网络中的真实状态。增量式计算的主要思想是差异化计算，在增量式计算的初始阶段，每次开始只用激活少量相较于上次发生改变的节点，从而可以达到快速收敛的效果。

和增量式计算模块主要交互的两个模块是主体存储模块的 TSGraph 和计算副本模块的 Spark RDD。每次 TSGraph 拿到要计算的任务首先会分解任务，一方面加载数据到计算副本模块，另一方面发送计算任务到增量计算模块。同时为了任务不至于丢失，保证任务的顺序执行，在 TSGraph 和增量式计算模块中间使用了 RabbitMQ，RabbitMQ 是一种消息队列。RabbitMQ 的作用一方面是 TSGraph 向 Spark 提交计算任务，另一方面是 Spark 完成任务的时候会返回任务完成消息的给 TSGraph，此时 TSGraph 会从计算副本抓取计算结果并和本地数据 Merge 后更新 HBase 中的本地数据。因为 TSGraph 提交任务的速度和 Spark 执行任务的速度不一定会匹配，Spark 返回任务完成的速度和 TSGraph 从计算副本同步数据的速度也可能不匹配，如果不使用消息队列的话会造成消息的覆盖，从而造成任务丢失或者是数据丢失。而采用消息队列的模式，可以保证任务一个个的执行，从而保证系统的容错性。

增量式算法模块还是用 Spark GraphX 的 API，实现了基于 BSP 的增量式算法，主要包括增量式的 PageRank，增量式的 TrustRank，增量是的 TunkRank 和增量式的 ShortestPatth。当接收到任务的时候，增量式计算模块会选择适合的计算方式保证计算的正常执行。增量式计算模块还有一个重要的功能是全量同步功能，增量式计算虽然可以带来计算效率的提升，但是同时也会造成结果准确度的算式，特别是在多次增量式计算之后结果的准确度会持续下降。此时需要全量计算来重新修正计算结果。修正计算结果的方法可以周期性的修复也可以按照增量数据的规模修复，具体可以由用户自己选择。

5.2.4 平台业务模块

平台业务模块主要是后端与前端的交互。为了实现一个易用的分布式图分析平台，我们使用 MVC 架构实现了一个业务平台。

平台业务模块主要包括 DataLoad Servixe，Runtime Service，Query Service 和 Analysis Service 几个模块。Dataload Service 主要负责数据的加载，OpenGraph 的数据模型和一般计算平台的数据模型不一样，特别是对于增量式计算的方式，需要增量式的加载数据，因此我们在这里写了一个单独的 DataLoad Service 模块，专门用来加载数据。Runtime 模块同样也是为了增量式计算服务，增量式计算是一个自动触发持续执行的过程，在这个执行过程中很少需要人工的干预，包括数据的增量式加载计算，数据的更新，结果的修正等都包含在 Runtime 模块中。Query 模块主要执行图上的一些查询操作，这个操作不涉及增量式计算的过程，主要是一个 OLTP 任务。Analysis 模块则是图上的集成分析模块，因为并不是所有的算法都适合增量式计算，有的计算还是得全量式计算才能得到准确的结果，Analysis 模块就是为了执行除了增量式计算之外的图计算任务。

5.3 OpenGraph 演示

如图所示就是 OpenGraph 动态图计算平台的首页，首页上首先有实时分析和历史分量两个导航栏。实时分析主要是执行增量式计算的任务，历史分析主要是为了计算图上的一些属性随着图演变的变化。

首先看实时分析页面，最上层是一个查询任务框。这个框的作用主要是实现一些基本查询的任务，查询过程中我们使用了 OpenCypher 这种基于图的查询语言。如图中所示，我们运行 match (n)-[r]-(t) return n,r,t 会返回整个图的结构，图中彩色节点部分就是我操作的大图。图中不同节点的大小代表不同节点的 PageRank，图中点的大小越大的节点，代表该节点的 PageRank 也比较大，表示该节点在整个网络中比较重要。图中不同颜色的节点代表不同的社区，由图中可以看出来同一个社区的节点关系比较复杂一点，不同社区的检点关系比较稀疏一

点。 该图会随着下层存储中节点数据的变化而变化，反映着整个网络的真实状态。

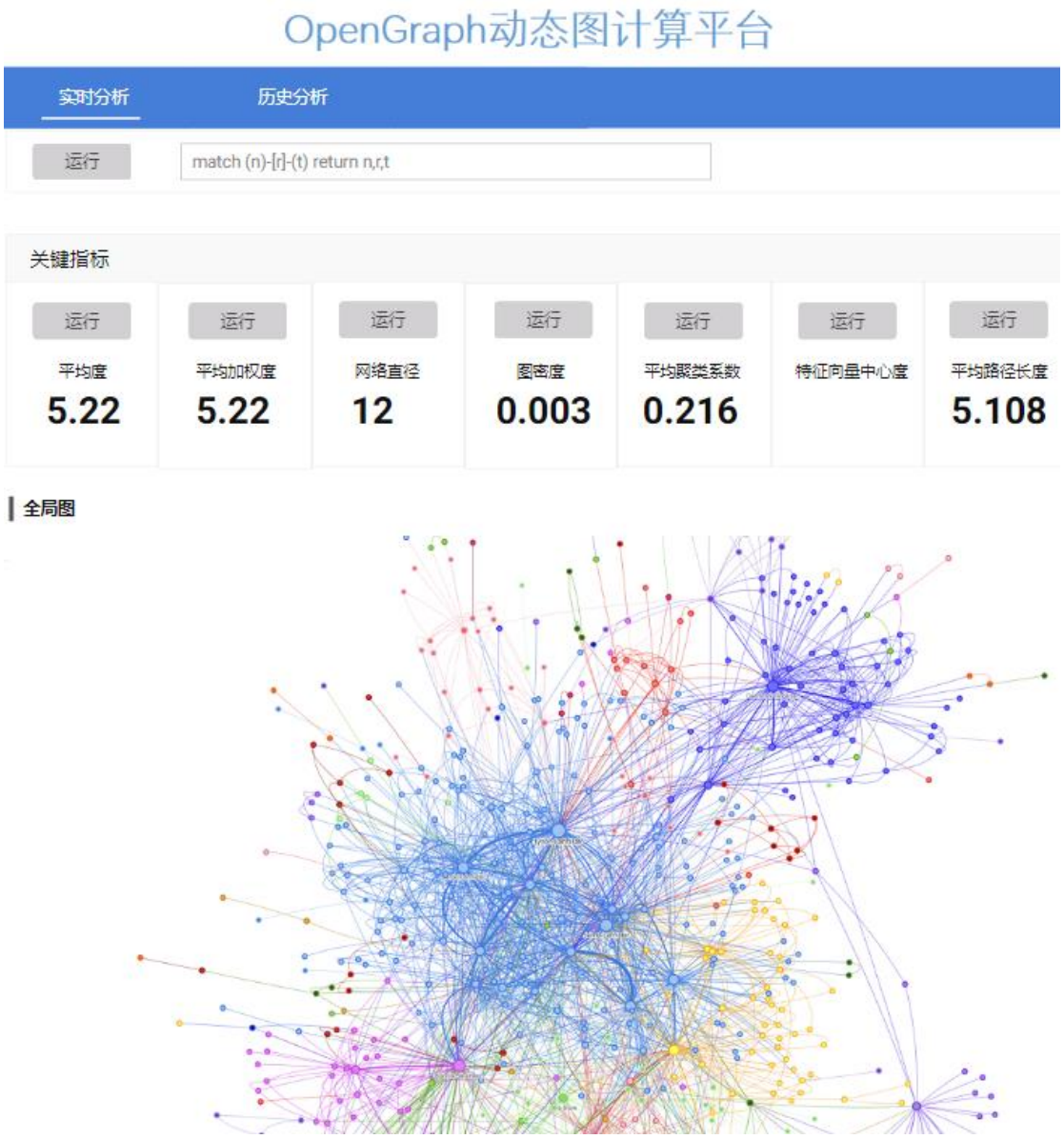


图 5-2 OpenGraph 图计算平台 DashBoard

在该页面上我们不仅展示了图的动态计算，同时也展现了一些其他图上信息的基本操作。这些操作包括一系列的基于图的属性统计和计算，首先是该图的平均度，平均度就是该图中所有节点的度的平均值。平均加权度指在计算平均度的基础上加了边权值的信息。网络直径表示任意两个节点之间的距离最大值。图密度是衡量图上的节点和边的比值的一种方式。平均聚类系数表示一个图形中节点聚集程度的系数。平局路径长度代表任意两个几点之间的平均最短距离。特征向量中心度表示图中节点在网络中的重要性程度。

这些算法都会根据用户的需要点击运行按钮而进行实时计算。例如平均度的计算，用户只需要点击平均度的运行按钮，系统就会生成图中所有节点的平均度。

并且会弹出一个新的窗口，新的窗口上有图中所有节点的度分布。可供用户详细查看。

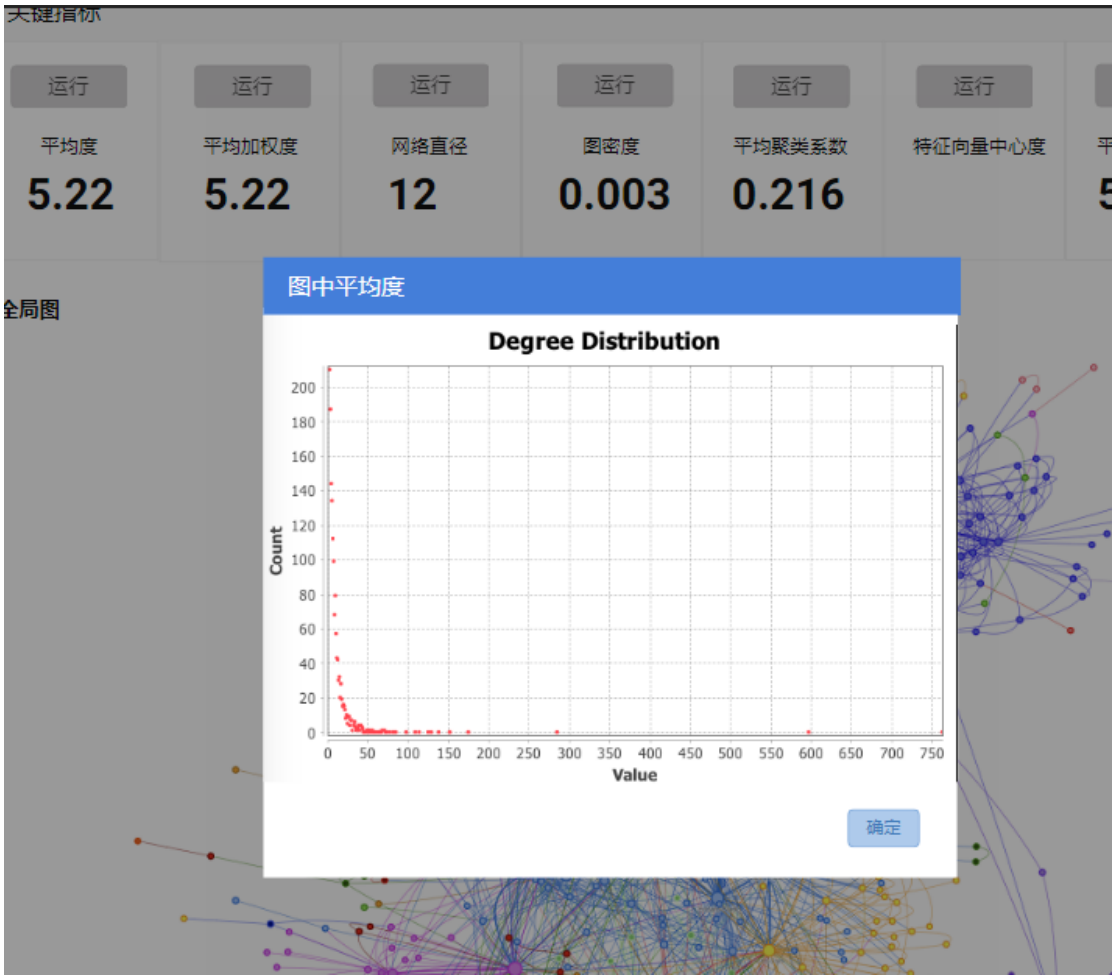


图 5-3 图属性分析页面

然后还有一个历史分析页面，因为我们的系统可以存一个动态图的历史信息。所以我们可以利用历史信息做一个图属性的历史分析。执行图属性的历史分析首先要选择历史分析的时间范围。

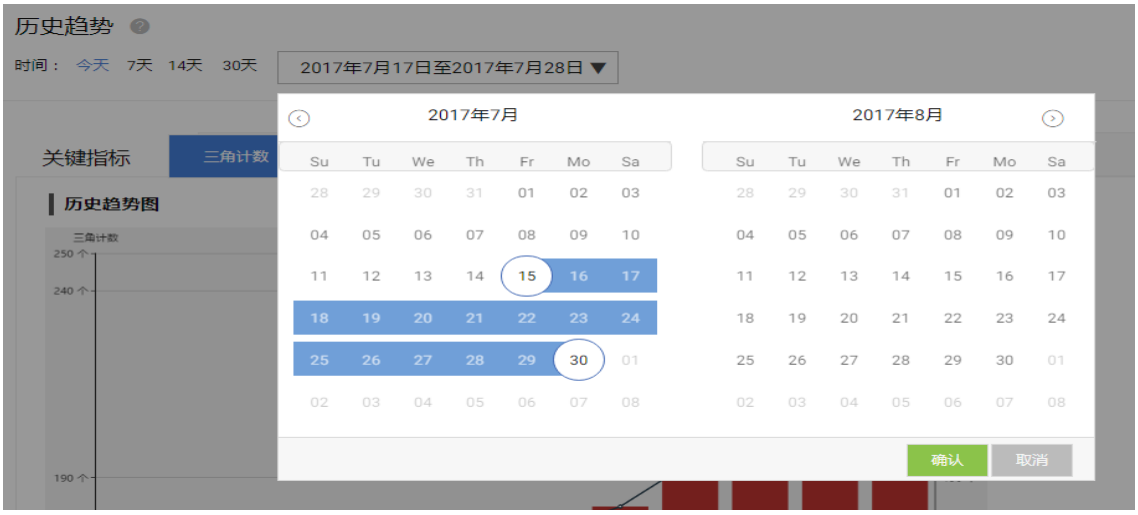


图 5-3 动态计算的时间区间选择

我们这里的历史分析主要以天为单位，主要是因为我们的数据更新并不是很快，以天为单位已经可以很好的反映数据的变化状态了。

图 5-3 历史计算结果展示

如图中所示，我们选择了 2017 年 7 月 17 日至 7 月 28 日的历史分析。主要针对三角计数属性进行分析。我们导入的数据是 Twitter 的评论数据，可以看出随着时间的增加三角计数的个数持续增加，证明了在这段时间内，该图的联系是越来越密切的。同时我们还支持紧密度中心性，结束中心性，连通分量，最大连通分量等属性的历史分析。大大的扩展了我们系统的适用范围。

5.4 小结

本章我们利用前几章的研究搭建了一个易用的动态图分析平台。第一节我们介绍了该平台的总体架构，围绕着平台的架构图做了功能模块的简单介绍。第二节我们从四个方面详细的介绍了该平台的设计理念，这四个模块包括时序图存储模块，计算副本模块，增量计算模块，平台业务模块。分别阐述了这四个模块之间的相互调用关系，分析了这样实现整个平台的优势。最后展示了该平台的前端交互页面。

总结与展望

随着互联网的飞速发展，原本相互独立或者关系稀少的数据联系越来越紧密，产生了大量带有复杂关系的数据。这些复杂的关系需要图分析技术才能准确的反映数据的真实状态。但是一方面现有的图计算技术都是对静态图的分析，然而现实世界中的数据模型都是时刻变化的，现有的静态计算模型不足以反映网络的真实状态。另一方面由于图的结构复杂，导致图计算的复杂度很难降低下来，分布式的成本也居高不下。这些原因一直制约着图分析方向的前景和图计算的发展。

本文中基于这两个关于图方面分析的痛点都提出了自己的方案，针对于现有的图分析都是基于静态图的痛点，我们设计了适合时序图存储的分布式图存储系统。通过存储结构的优化，查询索引的优化，可以在时间维度上快速查询到动态图的某个静态快照。针对现有的图计算算法复杂度居高不下的问题我们，设计实现了增量式图算法。通过实验表明我们的增量式图算法在一定的准确度丢失的容忍程度下，可以极大地提高图计算的执行效率。同时针对增量式算法会丢失准确度的这个问题，我们设计了全量校正的方案。可以保证我们的计算结果一直在某个准确度丢失的容忍范围内。

最后我们基于以上提到的动态图存储系统和增量式计算系统实现了一个支持动态图计算的综合图分析平台。该图分析平台主要有四个某块，包括数据的管理加载，图关系得查询，增量式算法的实时运行，全面的图属性分析指标。通过该分析平台，用户可以快速的对整个图有一个了解，并且可以根据自己的需求计算复杂的实时性任务。

虽然本文实现了一个完整的图分析平台，但是在一些地方仍然存在着优化空间。一个是在设计 TSGraph 的时候，没有考虑到更加复杂的情况，对读取和写入数据的安全性方面目前只加了读写锁，没有达到事务级别的支持，这对于一些复杂的操作可能会导致不一致的情况。另一方面关于增量式算法的研究，目前改造的增量式算法还比较少，下一步可以从更多算法的角度考虑，实现这些算法的增量化实现。

参考文献

[1]沈金萍. "第 39 次《中国互联网络发展状况统计报告》发布我国网民达 7.3 亿." 传媒 3(2017):30-30.
[2]Henzinger, Monika R. "Hyperlink analysis for the web." IEEE Internet computing 5.1 (2001): 45-50.
[3]Lin, Yankai, et al. "Learning Entity and Relation Embeddings for Knowledge Graph Completion." AAAI. 2015.
[4]Campbell, Karen E., Peter V. Marsden, and Jeanne S. Hurlbert. "Social resources and socioeconomic status." Social networks 8.1 (1986): 97-117.
[5]Lunt, Christopher, Nicholas Galbreath, and Jeffrey Winner. "Ranking search results based on the frequency of clicks on the search results by members of a social network who are within a predetermined degree of separation." U.S. Patent No. 7,788,260. 31 Aug. 2010.
[6]Papacharissi, Zizi, ed. A networked self: Identity, community, and culture on social network sites. Routledge, 2010.
[7]Givón, Talmy. Topic continuity in discourse. John Benjamins Publishing Company, 1983.
[8]H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media In WWW, pages 591-600, 2010.
[9]Broder A, Kumar R, Maghoul F, et al. Graph structure in the web[J]. Computer networks, 2000, 33(1): 309-320.
[10]Webber, Jim. "A programmatic introduction to neo4j." Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity. ACM, 2012.
[11]Shah, Seyyed M., et al. "A framework to benchmark NoSQL data stores for large-scale model persistence." International Conference on Model Driven Engineering Languages and Systems. Springer, Cham, 2014.
[12]Low, Yucheng, et al. "Distributed GraphLab: a framework for machine learning and data mining in the cloud." Proceedings of the VLDB Endowment 5.8 (2012): 716-727.
[13]Xin, Reynold S., et al. "Graphx: A resilient distributed graph system on spark." First International Workshop on Graph Data Management Experiences and Systems. ACM, 2013.
[14]Malak, Michael, and Robin East. Spark GraphX in action. Manning Publications Co., 2016.
[15]Malewicz, Grzegorz, et al. "Pregel: a system for large-scale graph processing." Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010.
[16]Chen, Rong, et al. "Powerlyra: Differentiated graph computation and partitioning on skewed graphs." Proceedings of the Tenth European Conference on Computer Systems. ACM, 2015.

[17]Spielmat, Daniel A., and Shang-Hua Teng. "Spectral partitioning works: Planar graphs and finite element meshes." Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on. IEEE, 1996.
[18]Abello, James, Frank Van Ham, and Neeraj Krishnan. "Ask-graphview: A large scale graph visualization system." IEEE transactions on visualization and computer graphics 12.5 (2006): 669-676.
[19]Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-113.
[20]Meissner, Andreas, et al. "Integrated mobile operations support for the construction industry: the COSMOS solution." Proceedings of of the 5th World Multi-Conference on Systemics, Cybernetics and Informatics, SCI. 2001.
[21]Shvachko, Konstantin, et al. "The hadoop distributed file system." Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on. IEEE, 2010.
[22]Zaharia, Matei, et al. "Apache Spark: A unified engine for big data processing." Communications of the ACM 59.11 (2016): 56-65.
[23]Justin Teller.Beringei: A high-performance time series storage engine.https://code.facebook.com/posts/952820474848503/beringei-a-high-performance-time-series-storage-engine.2017-02-03.
[24]Cheng, Raymond, et al. "Kineograph: taking the pulse of a fast-changing and connected world." Proceedings of the 7th ACM european conference on Computer Systems. ACM, 2012.
[25]苗又山. 大规模动态演化图的存储与分析系统研究. Diss. 中国科学技术大学, 2015.
[26]景年强, et al. "SpecGraph: 基于并发更新的分布式实时图计算模型." 计算机研究与发展 S1 (2014): 155-160.
[27]Rother, Carsten, Vladimir Kolmogorov, and Andrew Blake. "Grabcut: Interactive foreground extraction using iterated graph cuts." ACM transactions on graphics (TOG). Vol. 23. No. 3. ACM, 2004.
[28]Lee, Kisung, and Ling Liu. "Scaling queries over big RDF graphs with semantic hash partitioning." Proceedings of the VLDB Endowment 6.14 (2013): 1894-1905.
[29]Dhillon, Inderjit S., Yuqiang Guan, and Brian Kulis. "Weighted graph cuts without eigenvectors a multilevel approach." IEEE transactions on pattern analysis and machine intelligence 29.11 (2007).
[30]Vu, Nhat, and B. S. Manjunath. "Shape prior segmentation of multiple objects with graph cuts." Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on. IEEE, 2008.
[31]Kernighan, Brian W., and Shen Lin. "An efficient heuristic procedure for partitioning graphs." The Bell system technical journal 49.2 (1970): 291-307.
[32]Karypis, George, and Vipin Kumar. "METIS--unstructured graph partitioning and sparse matrix ordering system, version 2.0." (1995).
[33]Cheng, Jiefeng, et al. "VENUS: Vertex-centric streamlined graph computation on a single PC." Data Engineering (ICDE), 2015 IEEE 31st International Conference on. IEEE, 2015.
[34]Kyrola, Aapo, Guy E. Blelloch, and Carlos Guestrin. "Graphchi: Large-scale

graph computation on just a pc." USENIX, 2012.
[35]Valiant, Leslie G. "A bridging model for parallel computation." Communications of the ACM 33.8 (1990): 103-111.
[36]Chang, Fay, et al. "Bigtable: A distributed storage system for structured data." ACM Transactions on Computer Systems (TOCS) 26.2 (2008): 4.
[37]Beckett, S. "InfluxDB-archive/rollup/precision tuning feature, 2015." URL: https://github. com/influxdb/influxdb 1884: 38.
[38]TC Group. "Cacti: The complete rrdtool-based graphing solution." (2009).
[39]Wallace, Philip Richard. "The band theory of graphite." Physical Review 71.9 (1947): 622.
[40]Tuinstra, F., and J. Lo Koenig. "Raman spectrum of graphite." The Journal of Chemical Physics 53.3 (1970): 1126-1130.
[41]Stankovich, Sasha, et al. "Synthesis of graphene-based nanosheets via chemical reduction of exfoliated graphite oxide." carbon 45.7 (2007): 1558-1565.
[42]Agrawal, Bikash. Analysis of large time-series data in OpenTSDB. MS thesis. University of Stavanger, Norway, 2013.
[43]Faraj, Ihsan, et al. "An industry foundation classes Web-based collaborative construction computer environment: WISPER." Automation in construction 10.1 (2000): 79-99.
[44]Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.
[45]Page, Lawrence, et al. The PageRank citation ranking: Bringing order to the web. Stanford InfoLab, 1999.
[46]Gyöngyi, Zoltán, Hector Garcia-Molina, and Jan Pedersen. "Combating web spam with trustrank." Proceedings of the Thirtieth international conference on Very large data bases-Volume 30. VLDB Endowment, 2004.
[47] Das Sarma, Atish, et al. "A sketch-based distance oracle for web-scale graphs." Proceedings of the third ACM international conference on Web search and data mining. ACM, 2010.
[48]Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/index.html.

攻读学位期间取得的研究成果

[1] 赵炳,张雷.基于分布式计算的大规模动态图计算的研究[EB/OL].北京：中国科技论文在线[2018-01-10].http://www2.paper.edu.cn/releasepaper/content/201801-53.
[2]张帜，赵炳等. Neo4j 权威指南[M].北京:清华大学出版社,2017.