HybriG:一种高效处理大量重边的属性图存储架构

黄权隆[1)] 黄艳香[1)] 邵莹侠[1)] 孟 嘉[2)] 任鑫琦[2)] 崔 斌[1)] 冯是聪[2)]

[1)](北京大学信息科学技术学院高可信软件技术重点实验室 北京 100871)

[2)](北京明略软件系统有限公司 北京 102218)

摘 要 在图中,起点和终点都相同的两条边称为重边.属性图是一种带标志和重边的有向图,图中的点和边可以拥有任意数目的属性值.属性图由于其丰富的表达能力而广泛应用于实际建模中.实际应用中一般用图数据库解决属性图的存储需求.相比于传统的关系型数据库,图数据库在做多跳邻域查询、路径查询等与图结构相关的查询时,具有更优异的性能.Titan 是产业界日渐关注的一个开源的分布式图数据库,Titan 的数据以邻接表的方式组织,每个点的邻接表存储了相邻的所有边,这使得与邻接点集相关的查询都需要遍历整个邻接表.当图中含有大量重边时,邻接表规模巨大,这种数据组织方式导致邻域查询性能严重受损.邻域查询是大部分图查询的基础,如多跳邻域查询、路径查询、局部聚集系数查询(计算)等,这些查询往往由嵌套的邻域查询实现,随着邻域深度的增加,这种性能受损将被急剧放大.文中提出了一种基于 Titan 和列式存储数据库 HBase 的复合架构设计——HybriG,基于 Titan 和 HBase 建立存储层,用 Titan 来存储图的结构信息和点集的属性信息,HBase 存储边集的所有属性信息.在 HybriG 中邻接表保持了项数和数据量上的精简,从而能克服上述图数据库的缺点.相比于传统图数据库 Titan,HybriG 在邻域点集相关查询以及边集数据批量导入上的性能提升一个量级以上.文中介绍了 HybriG 基于 Titan 和 HBase 的存储设计,并描述了在此存储设计基础上,如何高效地实现图查询以及图数据的插入操作.此外,文中还提出了图数据的高效导入方案,并保证导入过程中 Titan 与 HBase 存储数据的一致性.最后通过实验验证了 HybriG 在处理大量重边时的优异性能.

关键词 属性图;重边;图数据库;Titan;HBase

中图法分类号 TP18 **DOI 号** 10.11897/SP.J.1016.2018.01766

HybriG: A Distributed Storage Architecture for Efficiently Processing Property Graph with Massive Multi-Edges

HUANG Quan-Long[1)] HUANG Yan-Xiang[1)] SHAO Ying-Xia[1)] MENG Jia[2)] REN Xin-Qi[2)]
CUI Bin[1)] FENG Shi-Cong[2)]

[1)](*Key Laboratory of High Confidence Software Technologies (MOE), EECS, Peking University, Beijing* 100871)

[2)](*MiningLamp Software System Co., Ltd. Beijing* 102218)

Abstract The past decades have witnessed the massive growth of Internet. Vast amount of graph data was produced by the boom of social network, e-commerce and online education. To analyze and manage graph data, there are two branches of development. One focuses on distributed graph computing, solving problems like PageRank or other algorithms that fit into the Bulk Synchronous Parallel (BSP) model. Systems like Pregel, GraphLab and PowerGraph are proposed for this branch. The other branch focuses on management of graph data, providing support like OLTP and graph queries. In this branch, graph databases like Neo4j and Titan are developed for

收稿日期:2016-12-13;在线出版日期:2017-04-24.本课题得到国家自然科学基金(61572039)、国家"九七三"重点基础研究发展规划项目基金(2014CB340405)和深圳政府研究项目(JCYJ20151014093505032)资助.**黄权隆**,男,1991 年生,硕士研究生,主要研究方向为数据仓库、分布式存储.E-mail:huang_quanlong@126.com.**黄艳香**,女,1990 年生,博士研究生,中国计算机学会(CCF)会员,主要研究方向为社交媒体数据分析、实时推荐.**邵莹侠**,男,1988 年生,博士,主要研究方向为大规模图数据分析.**孟 嘉**,男,1982 年生,硕士,主要研究方向为分布式存储.**任鑫琦**,男,1984 年生,硕士,主要研究方向为大数据计算和可视化.**崔 斌**,男,1975 年生,博士,教授,中国计算机学会(CCF)杰出会员,主要研究领域为数据库、社交数据分析、大规模图挖掘.**冯是聪**,男,1973 年生,博士,主要研究方向为大数据分析与挖掘.

management of property graph. The property graph is a directed, labeled graph with multi-edges, i. e., edges with the same source vertex and destination vertex. Vertices and edges can be associated with any number of properties. Since it can represent graph data in most scenarios, the property graph model has numerous applications in industry. However, traditional graph databases encounter significant performance degradation when the graph contains large amount of multi-edges. We reveal the cause of this in Titan. It's an open source distributed graph database, which has attracted wide attention in industry. Furthermore, we propose HybriG, a better distributed architecture based on Titan and HBase for this scenario. Titan stores graphs in adjacency list format, where a graph is stored as a collection of vertices with their adjacency lists. Each entry of the adjacency list stores an edge or a vertex property. When querying about adjacent vertices, Titan has to look through the entire adjacency list of the source vertex, which is a waste since we just need the connected vertices but not the multi-edges. This cost hurts the performance when the multi-edge set grows explosively. For high level queries that are based on adjacent vertices query, e. g. path queries, local cluster coefficient queries etc., the performance loss will be worse. HybriG implements property graph model as well. It stores the vertex data and graph structure in Titan, the edge data in HBase, respectively. We chose HBase since it's one of the storage engines of Titan and is widely used in industry. Storing part of the graph data into HBase won't bring too much cost because it's also the data store of Titan. This separation helps HybriG to keep a concise adjacency list about the graph structure, which helps to gain an order of magnitude improvement in execution of adjacent vertices based queries and batch loading of edge set. The difficulty of this separation solution is that we should guarantee the consistency of edge data between Titan and HBase. In most of the scenarios with large amount of multi-edges, multi-edges are used to represent event-like data, e. g. phone call records between two people, which won't be modified after insertion. Thus we can relax the consistency constrain of edge data to just guarantee consistency for insertion. We leverage the transaction in Titan to achieve consistency with HBase in edge insertion, especially in batch loading of edge data. Finally, extensive experiments have been conducted to show the outstanding performance of HybriG.

Keywords　property graph; multi-edge; graph database; Titan; HBase

1　引　言

图是一种表达能力丰富的数据结构，实际应用中往往用点表示实体，用边表示实体间的关系. 在有向图中，如果两条边具有相同的起点和终点，则称这样的边为重边(multi-edge). 属性图[1]是一种带重边的有向图，图中的每个元素(点/边)可以拥有任意数量的属性值. 特别地，每个元素都有一个 label 属性，标识该元素的类别.

实际应用中，有些属性图会含有大量重边. 比如在电信行业的通话关系图中，两人之间可以发生多次通话关系；又比如金融行业的交易关系图中，两人之间可以发生多次交易关系. 更复杂的，在刑侦应用中，整个情报系统获得的信息融合成一个知识图谱[2]，其中的实体和关系类型丰富多样. 比如实体类别有人、汽车、公司等，实体间的关系有亲属关系、共同出行关系、通话关系等. 有的关系是可以多次重复的，频次可能成百上千，这使得两个实体间不仅有多种 label(即多种关系)的边，还包含大量重边. 图 1 是

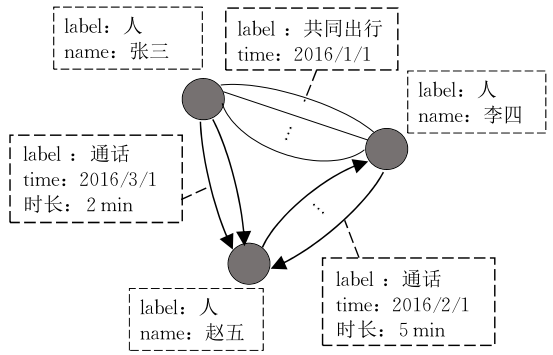

图 1　属性图示例

属性图在刑侦场景中的一个简单示例，显示了类型为人的结点以及通话关系和共同出行关系. 其中共同出行关系是双向的，因此在图中以无向边表示. 刑侦场景中的典型查询包括从一个人出发查询与其相关的所有人(邻域点集查询)、查询两个人在几跳内是否有关系(路径查询)、查询两人之间的所有关系(两点间边集查询)等.

由于重边在实际应用中常表示事件类型的关系(如一次通话记录)，传统解决方案将数据存储在关系型数据库中. 关系型数据库能胜任图数据的存储，也能方便地检索元素的内容，但在图结构相关的查询上表现欠佳. 比如在两跳邻域查询中，锁定一个嫌疑人后，要查看其两跳范围内的子图信息. 为得到第二跳的邻域结点，需要对各种关系表做昂贵的 JOIN 操作. 图数据库由于其以图的形式存储数据，在图的拓扑结构查询中具有更优异的性能. 因此当应用场景中图的拓扑结构查询与元素的内容查询同等重要时，图数据库是最佳的选择. 明略数据[①]的 SCOPA[②]平台即使用图数据库实现其存储核心. SCOPA 是明略数据的重要产品，在海量刑侦数据上构建起一个数据分析挖掘平台，展现给领域专家的是一张属性图，关联了客户的所有数据.

在富含重边的属性图中，图数据库的查询性能不佳. 图数据库以邻接表的方式存储图，表中每一项存储一条边. 这使得查询邻接点集时，需要遍历整个邻接表中的所有边，而查询目标只是这些边里邻接的点集数据. 当图中含有大量重边时，邻接表规模巨大，这种数据组织方式导致邻域查询性能严重受损. 邻域查询是大部分图查询的基础，如多跳邻域查询、路径查询、局部聚集系数查询(计算)等，这些查询往往由嵌套的邻域查询实现，随着邻域深度的增加，这种性能受损将被指数级放大.

一种简化重边的建模方式是，将所有同 label 的重边合并为一条边来表示，边上存储这些重边的所有信息，这样两个实体间的重边数能降低为边的 label 种类数. 然而，将多条重边的数据合并在一条边中存储，会使边的单位数据量骤增，图中的数据粒度过大，同时降低了单条重边的检索和插入效率，每次操作都要先读取相关的所有重边数据，再在其中做查询或插入操作. 因此，处理大量重边是一个不可避免的需求.

在许多富含重边的图应用场景中，数据的操作需求相对单一，而且没有很强的事务性要求. 比如在 SCOPA 的刑侦应用场景中，数据都是一些客观的事实数据，不需要修改，数据操作主要是图查询和批量的图数据插入. 图数据插入操作包括原始的全量数据导入和定期的增量数据导入，可以规避并发的写冲突，也不需要很强的事务性要求. 基于上述观察，在放宽了对强事务的支持后，我们可以设计一个相比传统图数据库更为高效的属性图处理系统，来应对含有大量重边的应用场景.

本文提出了一种基于图数据库 Titan[③] 和列式存储数据库 HBase[④] 的复合存储架构——HybriG. HybriG 能从容地处理富含重边的属性图，克服图数据库的局限性，具有更好的查询和插入性能. HybriG 架构应用于 SCOPA 的底层核心设计，在实践中也验证了其性能的优异性. 本文的贡献主要有：分析了图数据库处理大量重边时性能受损的原因，并基于此提出了解决方案 HybriG 及其存储、查询和高效数据导入设计，最后通过实验验证了 HybriG 的优异性能.

本文第 2 节介绍图数据库相关的预备知识，包括 HBase 及 Titan 的实现及其在处理大量重边时的局限性；第 3～6 节介绍 HybriG 的系统架构，包括查询模块的设计、数据高效导入方案的设计以及数据一致性的设计；第 7 节展示实验结果；第 8 节介绍大规模图数据处理的相关工作；最后在第 9 节对本文进行总结.

2 预备知识

2.1 HBase 简介

HBase 是 Hadoop[⑤] 生态系统中一个列式 NoSQL 数据库，基于 BigTable[3] 模型设计，搭建在 HDFS(Hadoop 分布式文件系统)之上.

BigTable 是 Google 在 2008 年提出的一个处理海量数据的 NoSQL 数据库. 图 2 是 BigTable 的模型示例. 在 BigTable 的模型中，数据表以行为单位组织，每行由一个键值唯一标识，称为行键(rowkey). 一行中可以包含任意数目的单元格(cell)，每个单元格由列名、版本号(时间戳)和值(value)组成. BigTable 中的行是以行键排序的，每一行中的单元格又是以列名和版本号进行排序，这使得其与关系数据库一样能快速定位到目标单元格. 区别于传统关系模型，BigTable 中的列名不需要事先定义，因为其底层实际为一个 Key-Value 存

① http://www.mininglamp.com 2017,3,29
② http://www.mininglamp.com/solutions/scopa 2017,3,29
③ Titan. http://titan.thinkaurelius.com 2017,3,29
④ Apache HBase. http://hbase.apache.org 2017,3,29
⑤ Hadoop. http://hadoop.apache.org 2017,3,29

储，如同数据结构 Map 不需要事先定义所有 key. 每个列名从属于一个列族，只有列族需要在定义表结构时给出.

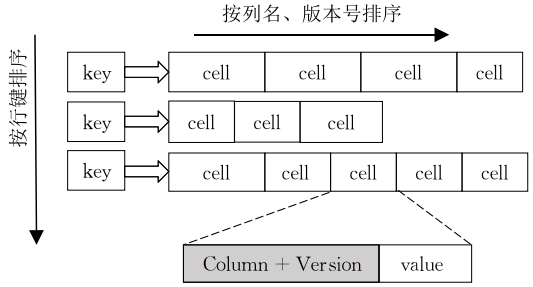

图 2　BigTable 模型示例

HBase 提供对某一行数据的 Put、Get、Delete 接口，能够具体操作该行中的给定列. 另外，HBase 提供给定行键范围的 Scan 接口，可以快速地获得连续几行的数据. Scan 接口也可指定特定的列或附加 Filter 来过滤无关数据.

HBase 由于其优异的可扩展性和稳定性，在产业界已得到广泛应用.

2.2　图数据库 Titan

图数据库是以图的形式来表示和管理数据的数据库[4]，与传统的关系型数据库相比，图数据库在图结构相关的查询上有更优异的性能，如多跳邻域查询、路径查询、局部聚集系数计算等.

Titan 是一个基于 Blueprints① 接口设计的开源图数据库，其实现了一个可插拔的存储接口，可以部署在 BerkeleyDB②、HBase 或 Cassandra[5] 之上. 相比于著名的图数据库 Neo4j[6]，Titan 是完全开源免费的，其受关注度正在与日俱增. 而且由于 Hadoop 生态系统在产业界的广泛应用，在 HBase 上搭建 Titan，即使用 Titan on HBase 应对图处理需求是较为常见的选择.

在图数据库 Titan 中，基于 BigTable 模型，数据在 HBase 中以邻接表的形式存储，如图 3 所示. Titan 为每个点分配了一个全局唯一的 id，每个点占 BigTable 中的一行，行键就是点的 id. 每行存储了该点相关的属性和边，它们各占一个单元格. Titan 在 BigTable 模型中设置最大版本数为 1，从而使每行中的列名与单元格一一对应，根据行键和列名可以快速定位到目标单元格，实现对边和属性内容的快速检索.

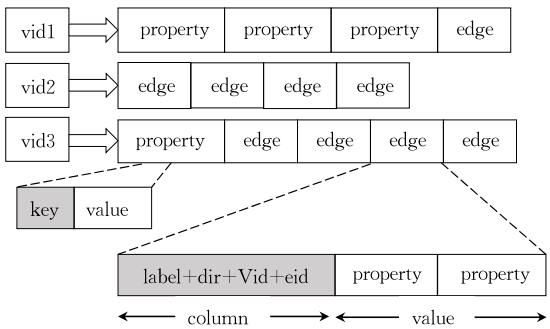

图 3　Titan 内部的 BigTable 实现

在 Titan 中，每个属性是一个 key-value 对. 点的属性存储在该点所在的行，每个属性占一个单元格，并以属性名 key 作为列名，这使得每个点的属性查询非常高效.

每个点所在的行还存储了邻接的所有边数据，每条边占一个单元格. 一条边的信息包含了邻接点、类别(label)、方向、边的唯一 id，以及边上的各属性. 单元格的值用来存储边上的所有属性，列名则存储边上除属性外的其它信息. 在 BigTable 模型中，同一行的单元格按列名排序. 为了方便检索，每条边所在的单元格依次以 label、方向、邻接点 id 以及边 id 拼接成为列名，即

列名 = 边 label + 方向 + adjVid + eid.

这种列名设计使得一个点的所有边先按 label 排序，再按方向、邻接点 id、边 id 等排序. 其优点是当要检索该点指定 label 的所有边时，只需要扫描邻接表的一部分. 图 4 是一个更详细的示例.

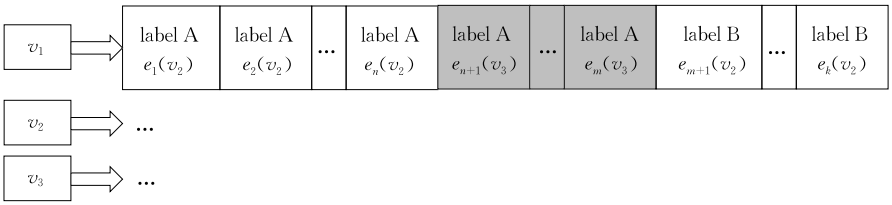

图 4　当属性图富含重边时，Titan 在 HBase 中的数据表是一张扁平而宽的表

2.3　图数据库 Titan 的局限性

当图中的重边数量巨大时，Titan 的邻接表列数也急剧变大，这会对邻域中点和边的检索带来严重影响. 图 4 展示了一个富含重边的场景，为方便展示省略了边的方向. v_1 只有 v_2 和 v_3 两个邻接点，其

① TinkerPop Blueprints: A Property Graph Model Interface. https://github.com/tinkerpop/blueprints 2017,3,29

② Oracle Berkeley DB Java Edition. http://www.oracle.com/technetwork/database/berkeleydb 2017,3,29

中与 v_2 有两种 label 的边，与 v_3 有一种 label 的边. 虽然邻接的点数不多，但 v_1 与邻接点都有数量巨大的重边. 由于邻接表存储的是每条边的信息，当要查询 v_1 的所有邻接点集（即 v_2、v_3）时，不得不遍历一次 v_1 的所有边，即遍历整个邻接表. 当重边数量巨大时，这是大量的无谓开销.

为了规避上述情形，我们可以换一种列名设计来优化邻域点集查询，比如让邻接表先按邻接点 *id* 排序，令：

列名＝adjVid＋边 label＋方向＋eid，

这样当发现一个邻接点时，可以跳过相同邻接点的所有边，不用再遍历整个邻接表. 然而，面对 label 相关的查询时又会面临同样的问题，比如查询该点总共有几种 label 的边，仍需要遍历该点的整个邻接表. 因此，改变邻接表的列名设计并不能解决问题，本质原因是邻接表存储了所有的边集.

另一方面，Titan 为加快数据访问设计了缓存，存储最近访问的点及其邻接表内容. 如果图中各点的重边都很多，则缓存空间被大量的重边占满. 由于是重边，这些边中只有为数不多的邻接点. 在做邻域点集相关查询时就会极大增加缓存的失效率.

面对富含重边的属性图，把所有边集都存在邻接表中是 Titan 性能受损的原因所在.

3 HybriG 的系统架构

为了更好地应对含有大量重边的属性图应用场景，规避传统图数据库的局限性，本文提出了一种复合存储架构 HybriG. HybriG 架构应用在明略数据的关联数据分析平台 SCOPA 中，提供属性图数据的查询与存储.

HybriG 将图的连接信息和点集的属性数据存储在 Titan 中，边集的所有属性数据则直接存储在 HBase 中. 由于 HybriG 架构基于 Titan 和 HBase 实现存储，而 Titan 的数据本身也存储在 HBase 中，为了方便表述以及避免混淆，下文中的 HBase 指的都是不包含 Titan 数据表的部分. Titan 在 HBase 中的数据表直接用 Titan 代称.

图 5 展示了 HybriG 的系统架构. Storage Layer 是 HybriG 的存储层，Query Adapter 和 Data Loader 分别为用户程序提供图查询和数据插入接口. 其中，Query Adapter 将图查询转换为对 Titan 和 HBase 数据的高效查询，再将结果汇总返回给用户程序. Data Loader 实现了数据的高效导入，在面对大批量的数据导入时，实现了错误恢复、断点恢复机制，同时保证了 Titan 和 HBase 的数据一致性. 下面将分别介绍 HybriG 各模块的实现.

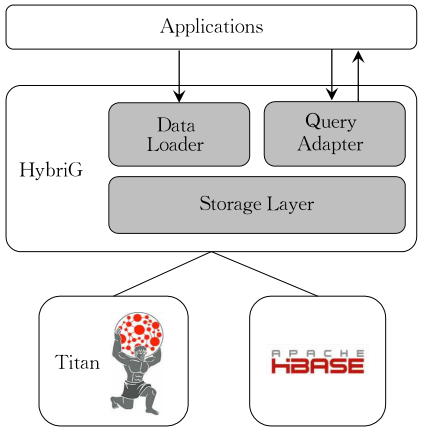

图 5 HybriG 系统架构

4 Storage Layer

HybriG 将图数据分开存储在 Titan 和 HBase 中. 具体地，如果两点之间有相同 label 的重边，HybriG 会在 Titan 中这两点间建立一条该 label 的边，将对应重边的数据都存储在 HBase 中的一个表，不妨称其为边表. 在 HBase 边表中，每条边占据一行，行键是 Titan 图中分配的边 *id* 拼接上重边的主键，单元格的值则存放具体边的内容. 重边的主键是这样定义的：在两点间同 label 的重边中，如果有一个属性能唯一确定一条边，则选该属性作为主键. 比如交易关系的时间戳，两个人在同一个时间点只能发生一次交易，因此该时间戳可以唯一确定两人间的一次交易. 如果某种 label 的边没有属性能唯一确定一条重边，则选该数据插入的时间戳作为主键.

在 HBase 边表中，同 label 的重边数据的行键都有相同的前缀（即 Titan 中的边 *id*），由于 HBase 中的数据是按行键排序的，它们在 HBase 中有相邻的位置. label 相同的重边经常会被同时检索，这种设计使得一次顺序扫描便可得到所有同 label 的重边. 另外，HybriG 利用 Titan 中边上的属性记录一些统计信息，如原图中实际有几条这样的重边，或者边上某个属性值的求和等，这些统计信息可以根据业务定制，用来加快业务相关的查询，不用再遍历相关的边.

图 6 是数据存储的一个示例，左边是实际要存储的图数据，e_1，e_2，…，e_n 都是 label 为“交易”的重

边，表示两人的所有交易记录. 右边是数据在 Titan 和 HBase 中的存储情况. Titan 中只存一条 label 为“交易”的边 $e_{1,2}$，该边有一个属性记录实际的边数. 利用这条边的 *id* 作为前缀，拼接上重边的主键（交易的时间戳）作为 HBase 的行键，将每条边的数据都存入 HBase 边表中对应的单元格中.

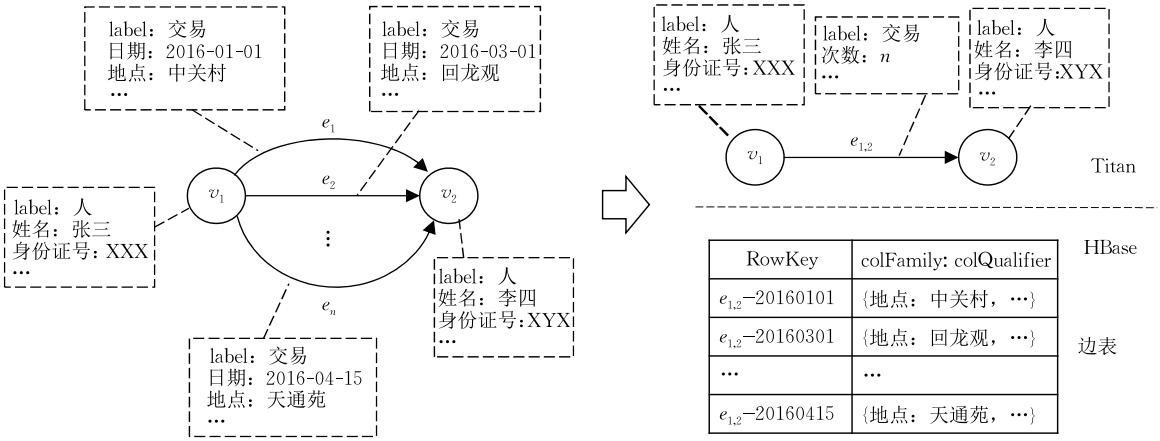

图 6　HybriG 存储示例（左图为原始数据的属性图，右图为数据在 Titan 和 HBase 边表的存储情况）

本文在引言中提到了一种避免产生重边的建模方式，即将所有同 label 的重边合并为一条边来表示，边上存储这些重边的所有数据. HybriG 架构与其有相似之处，但本质区别是边集元素的数据粒度仍是每条重边，不会将多条重边的数据作为一个单元来处理. 在 HybriG 中每条边的数据独占 HBase 边表中的一行，因此每条边的相关操作都会更加高效. 得益于 HBase 的高可扩展性，这种存储方式的性能也不会受制于数据规模的增长. 而前述建模方式将所有重边的数据合并在一条边中存储，当重边量级巨大时，对每条边的操作势必更加低效.

HybriG 采用的是合并同 label 的重边，而不是所有的重边，这仍是基于数据粒度的考虑. 图 7 是合并相同 label 重边后 Titan 的图示例（在 HBase 边表中的重边数据未显示）. 在实际应用场景中，经常需要根据具体的关系种类（label）来进行邻域点集查找，对于具体边数据的查询也常按 label 进行. 因此合并相同 label 的重边设计可以使很多查询在 Titan 中就得到满足，不需要再涉及 HBase 边表.

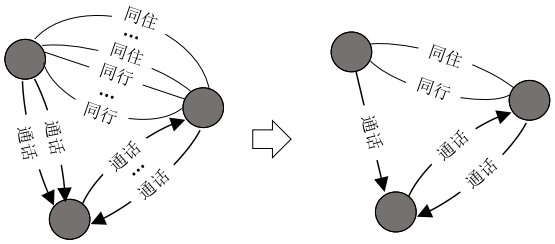

图 7　合并同 label 重边示意（左图为实际数据图，右图为 Titan 中的存储示意图. 无向边中的重边是指端点对应相同的两条或多条边，有向边中的重边是指起点和终点对应相同的两条或多条边）

5　Query Adapter

Query Adapter 模块将图查询转化为对应 Titan 和 HBase 的查询，再将结果汇总返回. 下面分别叙述其查询接口、查询实现以及查询优势.

5.1　查询接口

我们在 HybriG 架构上实现了基本图查询接口，暴露给上层应用的仍是一张属性图. 基本的图查询接口可分为如下几类：

（1）以点为中心的查询：获取某点邻接的点、获取某点的属性、获取某点邻接的边等；

（2）以边为中心的查询：获取某条边的端点、获取某条边的属性等；

（3）从图中直接查询：查询符合某种条件的元素（点/边），其中的条件可以是 label、限定返回集大小（limit）等.

更详细的属性图查询接口，可以查看 Blueprints 中的定义.

除了基本的属性图查询接口，HybriG 还提供了重边的统计信息查询. 比如查询两点间某种 label 的重边具体的边数、或者查询某个属性上的统计信息（如 MIN、MAX、AVG、SUM 等），可以根据业务需要来定制具体统计的信息. 比如通话关系所表示的边中包含了通话开始时间戳的属性. 可以设置 HybriG 统计该类重边在该属性的最小值，从而可以查询两人最早的通话发生时间. 在 HybriG 架构中，这些统计信息存储在 Titan 中的边上，因而可以

快速返回结果,无需再对具体的边数据(存在 HBase 边表中)进行统计.

区别于传统 Titan 图数据库,我们没有实现事务性的接口.因为业务场景可以规避对事务性的依赖,在富含重边的应用场景中,边集数据都是事件记录类型的数据,数据导入后就无需修改.而且数据导入可以分批定期执行,不会有多数据源写入导致的写-写冲突或读-写冲突.另外 HybriG 的数据分开存储在 Titan 和 HBase 两个数据库中,实现严格事务的代价比较大,会带来显著的性能牺牲.如 Google 的 Percolator[7] 在 BigTable 上实现了分布式事务,但写性能有 75%的牺牲.

5.2 查询实现

查询的实现可分为两部分,一部分只依赖于 Titan 中的数据,另一部分需要联合 Titan 和 HBase 边表的数据来返回结果.下面分别叙述.

5.2.1 仅依赖 Titan 数据的查询

在 HybriG 中,点集数据和邻接信息都存储在 Titan 中,因此只与点集相关的查询都可以直接转换为对 Titan 的查询,如点上属性的查询、查询给定点的邻域点集、在图中查询某种 label 的点等.

对于重边的统计信息查询,其需要的统计结果都已在 Titan 的边中存储,故可以直接转换为对 Titan 边上的属性值查询.具体实现中需要维护一个映射关系作为元信息,以得知一个统计信息对应 Titan 边上的哪个属性.

5.2.2 关联 Titan 和 HBase 数据的查询

当查询具体的边数据时,就需要关联 Titan 和 HBase 来实现查询.在 HybriG 架构中:

HybriG_Edge=TitanEdgeID+HBaseRowData.

上式表示每条边对象的两部分组成.所有边集数据存储在 HBase 的边表中,每条边的数据占据一行,存储其所有属性,该行的行键是 Titan 中对应的边 *id* 拼接上该边的主键值.因此查询某条重边的数据,需要先在 Titan 中找到对应边的 *id*,再在 HBase 边表中找到对应行的数据.

下面以两点间边集查询为例阐述 HybriG 的实现.两点间边集查询是指给定两个点,查询它们之间满足某种条件的边.比如已经从邻域查询得知 v_1 与 v_2 相邻,现在想得到它们之间某种 label 的所有边.查询的伪代码见算法 1.

算法 1. 两点间给定 label 的边集数据查询.

输入:点 v_1,点 v_2,*eLabel*

输出:两点间的所有符合条件的边集数据

```
1. e=v1.query().adjacent(v2).label(eLabel)
       .limit(1).edges().next()
2. IF e==null THEN
3.     RETURN null
4. END IF
5. res=hbaseTable.scan(e.id, next(e.id))
6. RETURN wrapEdges(res)
```

算法 1 中,第 1 行查询两点间该 label 的边,并利用 *limit*(1)修饰来加速.在 HybriG 架构中,Titan 只在这两点间存储该 label 的一条边,因此我们可以利用边的唯一性来加速查询.第 2~4 行,若在 Titan 中查询到该 label 的边不存在,则不需要再在 HBase 中检索.第 5 行根据 Titan 中的边 *id*,在 HBase 的边表中通过 Scan 查询得到所有边的具体数据.*e.id* 是一个字符串,伪代码中的 *next*(*e.id*)表示同等长度的下一个字符串,即将字符串 *e.id* 中最后一个字符的值加一.比如 *e.id* 为"abbb",则 *next*(*e.id*)为"abbc".HBase 的 Scan 函数返回连续的若干行数据,接收的两个参数是行键范围的起始和结束点,是一个左开右闭的区间.用 *e.id* 和 *next*(*e.id*)作为该区间的左右端点,即可查询到所有行键以 *e.id* 作为前缀的数据.最后第 6 行 *wrapEdges* 函数将这些数据转换为具体的边对象,每行一条边.

5.3 HybriG 的查询优势

5.3.1 邻域点集相关查询的优势

为了解释 HybriG 在图查询上的优势,图 8 展示了 HybriG 在 HBase 中的表结构,包括 Titan 自

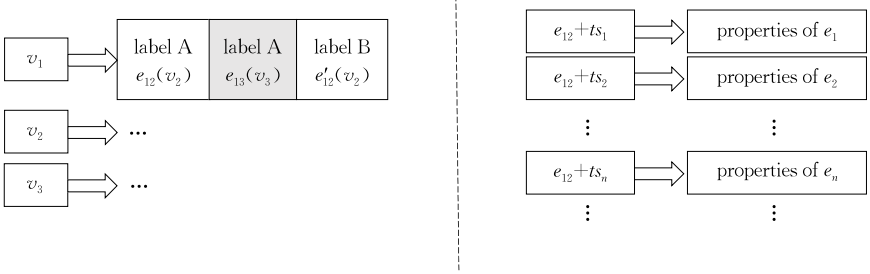

图 8 HybriG 在 HBase 中的数据表分为两张(左边为 Titan 转化成的数据表,右边为边表.当属性图富含重边时,Titan 数据表不被影响,边表将是一张高而窄的表)

身的 HBase 数据表和 HybriG 特殊设计的边表. 当属性图的重边数量爆炸性增长时，Titan 的数据表并不会被影响，因为两点间同 label 的边至多只会存在一条，每个点的邻接表列数仍为不同 label 邻接的点数. 另外，Titan 中的边只存放统计信息，因此每个单元格(即每条边)的数据量实际很小. 面对含有大量重边的属性图，各点的邻接表仍保持了列数和数据量上的精简. 即使面对邻域点集查询的全表扫描，也能提供优异的性能.

另一方面，得益于邻接表的精简，Titan 的缓存因此可以存放更多点集数据. 对于邻域点集相关查询，如多跳邻域(k-hop)查询、路径查询、局部聚集系数计算等，具体的实现往往由多个基本的邻域点集查询组成，缓存的命中率就显得尤为重要. 传统的解决方案中使用 Titan 存储所有的重边，使得缓存中只能存储少量点集的数据，大部分空间被边集数据占据，而具体的边集数据在查询中并不相关. 在 HybriG 架构中，Titan 的缓存能充分保留更多的点集数据，从而进一步提高邻域点集相关查询的缓存命中率.

综上，HybriG 对邻域点集相关的查询具有很好的表现，后续的实验结果将展示具体的数据.

5.3.2　边集相关查询的优势

在许多刑侦推演场景中，往往只需要查询两点间拥是否拥有某种 label 的边，而不需要查看具体各个重边的数据. 比如得知两人之间有表示共同住宿酒店的边相连后，基本可以断定两人认识，领域专家可以在图中继续推演出其他相关的人，后续有需要再展开这条关系，查看具体的各条酒店住宿记录. HybriG 架构为这种场景提供了便利，上述场景相当于在 HybriG 架构的 Titan 图中进行游走(Graph traversal)，当有需要时再展开某条边，在 HBase 边表中读回相应的边数据.

对于边集数据的读取，HybriG 将所有边的数据存储在 HBase 边表中，而且每条边占一行，这使得对边的检索是行级别的检索，即在表中查找一行. 传统的解决方案将重边数据都存储在 Titan 中，边集数据存储于各点的邻接表，而每个点的邻接表在 HBase 数据表中占据一行，因此对边的检索是选定行后的列级别检索. HBase 中行级别的检索要略优于列级别的检索，因此 HybriG 会略占优势. 然而，HybriG 对边的检索需要跨 Titan 和 HBase 两个系统，会多一次交互的开销. 实验表明，这两方面功过相抵，HybriG 的边集数据查询性能与 Titan 相差不大.

HybriG 架构在边集统计信息的查询或计算上会有很大优势. 在 HybriG 架构中，Titan 中某种 label 的边如果存在，则代表原属性图中两点间拥有该 label 的边. 具体的重边数据存储在 HBase 中的边表，而 Titan 中的边上的则存储了该 label 声明时设定的统计信息，如具体的重边数目、边上某个属性的最大最小值等. 对于这些统计信息，可以直接在 Titan 中查询得到结果.

6　Data Loader

Data Loader 模块负责图数据的导入，包括点的导入和边的导入. 在 HybriG 架构中，点集数据都存储在 Titan 中，故点集数据导入直接在 Titan 上完成. 边集数据虽然存储在 HBase 中，但 Titan 中的对应边上存储了重边的统计信息，因此需要同时更新 Titan 和 HBase，以保证二者数据的一致. 下面详述边的数据导入，以及如何保证 Titan 和 HBase 的数据一致性.

6.1　边数据的高效导入

在 HybriG 架构中，边的插入既要更新 Titan，又要更新 HBase 边表. 对于给定 label 的一条边数据，首先要查看 Titan 中两点间是否已有一条边，表示这样的关系存在. 若这样的边不存在，则将其创建. 然后更新 Titan 边上的统计信息，再利用其边 id 将边的属性数据存入 HBase 中. 算法 2 展示了上述逻辑. 第 1 行查询 Titan 中该 label 的边，由于至多只有一条，故可用 $limit(1)$ 加速. 第 2～4 行若这样的边不存在，则在 Titan 中添加一条边. 第 5 行根据待插入的边数据来更新 Titan 中边上的统计信息. 第 6 行提交对 Titan 的所有修改. 第 7 行将边数据写入 HBase 边表中，行键为 $e.id$ 拼接上边的主键.

算法 2.　插入一条边的伪代码.

输入：Titan 图接口 $graph$，HBase 边表接口 $hbaseTable$，点 v_1，点 v_2，$edgeLabel$，边数据 $realEdge$

```
1. e=v1.query().adjacent(v2).label(edgeLabel)
       .limit(1).edges().next()
2. IF e==null THEN
3.     e=v1.addEdge(v2, edgeLabel)
4. END IF
5. updateStats(e, realEdge.properties)
6. graph.commit()
7. hbaseTable.put(e.id+realEdge.primaryKey,
       realEdge.properties)
```

传统的解决方案将所有重边存储在 Titan 中，边数据的导入逻辑只需要上述代码的第 3、5、6 行. HybriG 架构中每条边的插入多加了一次查询操作(第 1 行)，以及 HBase 边表的插入操作(第 7 行)，带来了额外的时间开销. 而且最耗时的也是这两步操作，分别要从底层存储读取数据以及持久化所有修改到底层存储中. 如果有批量重边需要导入，这些开销可以让所有重边来平摊，即在数据导入时，对于两点间同 label 的重边作为一批来处理，这样就可以共享查询操作的结果，对 Titan 边上统计信息更新的持久化操作(commit)也只需进行一次. 批量导入重边数据的伪代码如算法 3 所示.

算法 3. 重边数据的批量导入.

输入：Titan 图接口 *graph*，HBase 边表接口 *hbaseTable*，点 v_1，点 v_2，*edgeLabel*，重边数据集 *allRealEdges*

```
1. e=v1.query().adjacent(v2).label(edgeLabel)
     .limit(1).edges().next()
2. IF e==null THEN
3.   e=v1.addEdge(v2, edgeLabel)
4. END IF
5. FOR realEdge IN allRealEdges DO
6.   updateStats(e, realEdge.properties)
7. END FOR
8. graph.commit()
9. FOR edge IN allRealEdges DO
10.  hbaseTable.put(e.id+edge.pk, edge.properties)
11. END FOR
```

在算法 3 中，第 1 到 8 行是对 Titan 的更新，将两点间同 label 的重边作为一批处理，共享了对 Titan 的操作，从而平摊了额外的开销. 另外第 9 到 11 行是对 HBase 边表的多次 *put* 操作，可以使用 HBase 的 Bulk Loading① 技术来进行加速.

6.2 数据一致性

在 Titan 的接口设计中，对图(*graph*)的初次操作将自动打开一个事务，执行 *graph.commit*()时该事务提交，将事务里的修改持久化到底层存储中，Titan 的事务保证了自身数据的一致性. HybriG 架构由于把边数据的存储分开在 Titan 和 HBase 上，需要保证 Titan 和 HBase 上数据的一致性. 比如 Titan 上关于重边的统计信息跟 HBase 边表的一致性，或者 HBase 边表里各行行键的边 *id* 部分跟 Titan 里的一致性. 由于实现强一致性的代价过高，本架构保证的是 Titan 和 HBase 数据的最终一致性[8]，即系统保证在经过错误恢复后，数据最终会达到一致的状态. 最终一致性足以满足应用场景的需求.

由于只有边集数据是跨 Titan 和 HBase 存储的，下面讨论的都是边数据的导入. 图 9 演示了边数据导入的 3 个步骤，第①步对应前述算法 3 中的第 1～7 行，更新 Titan 数据；第②步对应第 8 行，提交修改；第③步对应第 9～11 行，利用边 *id* 将边数据插入 HBase 边表中. 对数据的持久化修改是第②③步，数据会出现不一致的原因是②③并不是原子的. 如果②成功但是③失败了，即成功持久化了对 Titan 数据的修改，但对 HBase 边表的修改却失败了，则两边的数据出现不一致. 失败的原因是多种多样的，比如程序内存溢出(Out of Memory)、网络中断、硬件故障等.

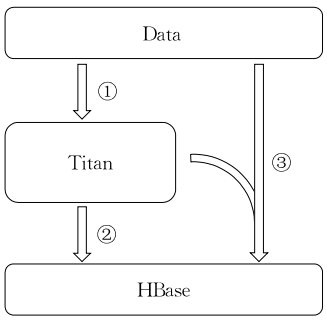

图 9 边的数据导入分 3 步：①更新 Titan 数据；②提交 Titan 更改即 *graph.commit*()；③将数据插入 HBase 边表中

当数据导入程序在故障之后重启时，这种不一致性需要被修复. 该批次的边数据将会被重新导入，为了得知②是否已经成功，我们需要知道 Titan 中的数据是否已经被修改了. 这可以利用 Titan 的事务原子性来实现：在②提交之前，往 Titan 中写入一个成功标志(可以用一个点或属性来表示)，然后再提交. 根据原子性，该标志被成功写入当且仅当 Titan 的更新都被持久化. 这样当数据导入程序重启时，我们先查看该标志是否存在，便可得知②是否已经成功了. 若其已成功，则跳过这一步，直接进行第③步. 该过程的伪代码如算法 4 所示.

算法 4. 保证一致性的边集数据批量导入.

输入：*graph*，*batchID*，HBase 边表 *hbaseTable*，点 v_1，点 v_2，*edgeLabel*，*allRealEdges*，所有边的属性

```
1. e=v1.query().adjacent(v2).label(edgeLabel)
     .limit(1).edges().next()
2. IF hasNotCompleted(graph, batchID) THEN
3.   IF e==null THEN
4.     e=v1.addEdge(v2, edgeLabel)
```

① http://hbase.apache.org/book.html#arch.bulk.load

```
5.    END IF
6.    FOR realEdge IN allRealEdges DO
7.      //按需更新边上的统计信息
8.    END FOR
9.    markCompleted(graph,batchID)
10.   graph.commit()
11. END IF
12. FOR edge IN allRealEdges DO
13.   hbaseTable.put(e.id+edge.pk, properties)
14. END FOR
```

算法 4 中，第 2 行的函数 *hasNotCompleted* 判断给定的 Titan 图中是否存在给定批次的成功标志. 若不存在则执行第 3 到 10 行更新 Titan，其中第 9 行的 *markCompleted* 函数在 Titan 图中插入该批次的成功标志.

值得一提的是，我们在 HBase 中并没有放置成功标志. 如果数据导入程序在第③步成功插入 HBase 数据后出现故障，则重启后还会将边集数据再次插入 HBase 中. 但这是没有问题的，因为 HBase 边表中的数据不存在统计信息，因此不存在更新操作，所有操作都是新数据的插入操作. 我们设置 HBase 的最大版本数为 1，则多次插入同一内容到一个单元格，实际只保存一份，不会增加存储开销.

7　实验和分析

在现有的公开数据集（比如 LDBC[①]、SNAP[②]、LAW[③]）中，并没有富含重边的场景，这些图也不是属性图. 富含重边的属性图普通存在于电信、金融、刑侦等行业中，数据是不公开的. 由于实际的数据只能在相关部门内部查询，明略数据根据客户数据中统计出的特征构造测试用图，以支持 SCOPA 的开发与测试，验证 HybriG 架构的优秀性能.

本组实验选用的图中有 20 万个顶点，48 197 700 条边，平均度数为 482，每个点平均与 3.6 个点相邻，两点间同 label 的重边的平均重数为 134. 实验对比的是直接将图存储在 Titan 中的传统方案以及将图存储在 HybriG 中的方案.

实验环境为 5 台服务器组成的集群，每台机器安装 Ubuntu14.04 操作系统，物理配置均为一个 Intel Xeon E3-1220（3.10 GHz）处理器、16 GB 内存、一个 1 Gbps 网卡及一个 4 TB SATA 接口硬盘. HBase 部署在这 5 台机器上，每台机器的 RegionServer 设置最大堆内存为 4 GB. 其中 HBase 版本为 1.0.1.1，Titan 版本为 0.5.4.

7.1　邻域点集相关查询

邻域点集查询是指查询给定点的邻接点集. 许多图查询基于邻域点集查询实现，如 k-hop 点集查询、局部聚集系数查询、广度优先搜索等. 下面叙述两个基于邻域点集查询的图查询实验.

7.1.1　k-hop 点集查询

k-hop 点集查询即查询给定点在 k 跳能到达的点集. 实验在测试图中随机抽取 100 个点作为起点，查询它们的多跳邻域点集，测试它们的平均查询时间. 同时又对小邻域（邻域点数小于 5）的点集和大邻域（邻域点数大于 20）的点集进行了同样的实验. 表 1～表 3 是 Titan 与 HybriG 的性能表现.

表 1　随机选 100 个点的 k-hop 查询平均耗时

hops	架构	
	Titan/ms	HybriG/ms
1	13.65	4.25
2	72.97	18.73
3	446.34	34.69
4	2755.86	99.78

表 2　随机选 100 个小邻域点的 k-hop 查询平均耗时

hops	架构	
	Titan/ms	HybriG/ms
1	8.67	2.08
2	31.08	7.79
3	179.47	19.88
4	1140.00	52.45

表 3　随机选 100 个大邻域点的 k-hop 查询平均耗时

hops	架构	
	Titan/ms	HybriG/ms
1	17.79	3.52
2	110.34	14.49
3	721.25	39.98
4	4781.79	163.15

随着跳数的增加，两系统的查询时间都呈指数增长. HybriG 不管在 1 跳的初始值还是在增长的倍数上都远优于 Titan. 正如 2.3 节分析的，对某个点的邻域点集查询需要遍历该点的邻接表. 当图中的重边数目巨大时，Titan 受累于其庞大的邻接表，对邻域点集的查询速度大大降低. 而 HybriG 由于极大简化了存储在 Titan 中的边数目，使得邻接表的数据量大大缩小，从而降低了邻接表的遍历耗时.

7.1.2　局部聚集系数计算

局部聚集系数（Local Cluster Coefficient）是以

① LDBC. http://ldbcouncil.org
② SNAP. http://snap.stanford.edu/data
③ Laboratory for Web Algorithmics. http://law.di.unimi.it/datasets.php

点为中心的度量，表示其邻域子图的紧密程度，即邻居节点之间有多大比例有边相连. 定义如下：

$$LCC(v)=\frac{|\{(u,w):u,w\in N_v,e(u,w)\in E\}|}{|\{(u,w):u,w\in N_v\}|},$$

其中 N_v 为点 v 的邻域点集，E 为图中的边集. 上式表示邻域的点对集合中，有边相连的点对比例. 由于 $e(u,w)\in E$ 等价于 $w\in N_u$，分子部分可转化为

$$|\{(u,w):u,w\in N_v,w\in N_u\}|=\sum_{u\in N_v}|N_u\cap N_v|.$$

从上式可知，局部聚集系数的计算需要对邻域点集中的点再做一次邻域点集查询，因此其复杂度相当于 2 跳邻域点集查询.

实验对比的是 HybriG 与直接将图存储在 Titan 中的传统方案. 测试点集的抽取方式同 7.1 节，即小邻域(邻域点数小于 5)的点集、随机采样点集和大邻域(邻域点数大于 20)的点集，每个点集拥有 100 个点. 图 10 是实验结果.

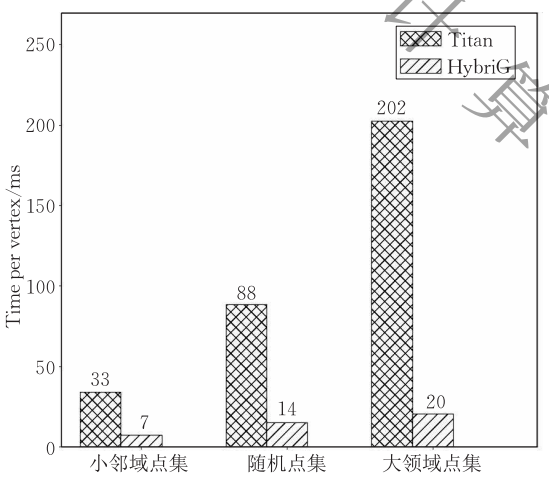

图 10 计算 100 个点的局部聚集系数的平均耗时

与 k-hop 查询一样，HybriG 在局部聚集系数计算的性能上要远优与 Titan，而且在大邻域点集上的优势更加明显.

7.2 边集相关查询

下面介绍边集相关查询的两个实验结果. 7.2.1 节是两点间边集查询，在刑侦场景中，图中的一类顶点代表人，人之间的边代表两人的关系，比如共同出行记录、共同住宿酒店记录、通话记录等. 当研判专家锁定两个嫌疑人后，需要查询两人间的关系数据，即为两点间边集查询. 7.2.2 节是邻接边集查询，在刑侦场景中，锁定嫌疑人后要查看其某种类型的关系数据，即为邻接边集查询. 这两种查询的图查询语句是不同的，前者限定了边的两个邻接点，后者只给定了边一个端点，侧重于限定 label. 使用 Blueprints 图查询接口，两点间边集查询的示例语句如下：

$$v_1.query().adjacent(v_2).edges().$$

邻接边集查询的示例语句如下：

$$v.getEdges(Direction.BOTH,eLabel).$$

其中 v_1、v_2、v 是顶点，$eLabel$ 是边的一种 label，$Direction.BOTH$ 是常量，代表边的方向可以是入边或出边.

7.2.1 两点间边集查询

实验测试的是给定两个点和一个 label，查询两点间该 label 的所有边. 在图中随机抽取 100 个有边相连的点对，实验测试查询这些边的耗时.

图 11 统计了两种系统的时间对比，两种方案的查询耗时非常接近. HybriG 的存储层基于 Titan 和 HBase 实现，对边集的查询先要在 Titan 里查询边 id，再在 HBase 的边表中查询具体的边，因此时间开销分两部分(详见 5.1 节). 图中展示了 HybriG 查询时间的两部分组成. 两种方案的耗时相近主要有两方面的原因.

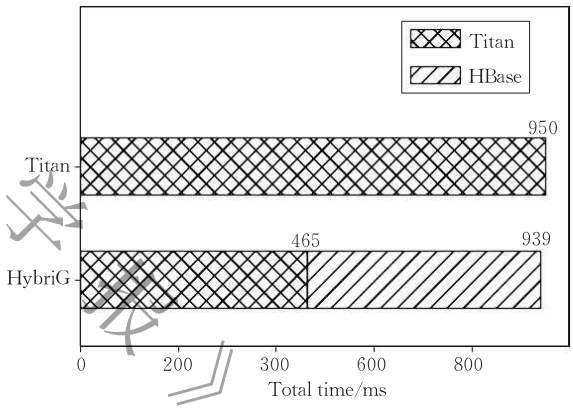

图 11 查询 100 个点对间所有边的总耗时

一方面，HybriG 检索边集数据需要在 Titan 和 HBase 这两个数据库中进行查询，且这两个查询不能并行. 传统方案只需要在 Titan 中查询，HybriG 多加了一轮查询的时间开销. 然而，这部分的时间开销并不是很大. Titan 的数据表本身也存储在 HBase 中，HybriG 的查询实际上只是 HBase 中的跨表查询. 跨表查询能共用 HBase 的一些缓存信息，如 HMaster、RegionServer 的位置信息、HBase 中 Root 表和 Meta 表的信息等. 不过尽管只是跨表查询，在这方面 HybriG 还是引入了时间开销.

但另一方面，HybriG 对边集的查询是转化为 HBase 边表中的行级别检索(详见 5.1 节). 而将图存储在 Titan 中的方案，对边集的查询实际是转化为 HBase 数据表中选定行后的列级别检索. 当图中

含有大量重边时，前者是在一张高窄(tall-narrow)表中查找连续的几行，后者是在一张扁宽(flat-wide)表中选定一行后查找连续的几列，前者的性能会略优一些.因此在这方面 HybriG 略优.

7.2.2　邻接边集查询

实验测试的是给定一个点，查询其某种 label 的所有边.点集的抽取方式同 7.1 节，即小邻域(邻域点数小于 5)的点集、随机采样点集和大邻域(邻域点数大于 20)的点集，每种点集抽取 1000 个点.

图 12 展示的是两种系统的查询时间，以及查询耗时中具体的时间组成，其中 HybriG 的查询时间分为 Titan 中的查询耗时和 HBase 中的查询耗时两部分.当边集规模增大时，HybriG 在 Titan 中的查询耗时并没有显著增加，而在 HBase 边表中的查询耗时增长的幅度也没有传统方案中的 Titan 大.

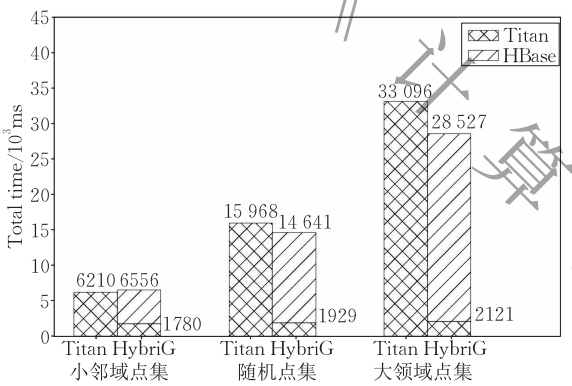

图 12　查询 1000 个点给定 label 的所有边总耗时

7.3　边数据的导入速度

关于边的数据导入速度，我们对不同规模的图进行了测试.对比直接将图存储在 Titan 以及将图存储在 HybriG 的两种方案.我们基于 MapReduce[9] 开发了分布式的导入程序.表 4 是边数据导入时间的统计.

表 4　边集数据的批量导入耗时

测试图号	点数	边数	重边的重数	Titan 导入时间	HyBriG 导入时间
1	50 000	23 786 800	100	30 min	14 min
2	200 000	48 197 700	100	45 min	16 min
3	200 000	481 977 000	1000	9 h 26 min	1 h 24 min

可以看到，在所有测试用图中，HybriG 对边集数据的导入拥有更高的速度.这主要有两方面的原因：一是 HybriG 对 HBase 边表的数据导入使用了 HBase 的 Bulk Loading 技术.二是 Titan 对新增的点和边都会分配一个全局唯一的 *id*(这也是 Titan 没有实现 HBase Bulk Loading 的原因)，HybriG 大大减小了自身 Titan 中的边数目，从而节省了 *id* 分配的时间开销.

8　相关工作

图数据分析与处理是大数据背景下的一大应用分支[10]，大数据背景下的图处理系统可以分为图计算框架和图数据库两大类.

图计算框架旨在高效地进行全图量级的并行计算，如计算 PageRank[11]、社区发现[12](Community Detection)、子图匹配[13] 等，提供的是对 OLAP (Online Analytical Processing)的支持.这类系统中，Pregel[14]、GraphLab[15]、PowerGraph[16] 支持 BSP[17] 模型的计算，对图数据进行特定的分割，使得集群中的机器可以在内存中计算分区的数据，再通过交互完成全量数据的计算.上述系统中，Pregel 没有开源，GraphLab 和 PowerGraph 都需要单独部署，无法有机地融入已有的大数据生态系统中，需要导出数据以供计算.GraphX[18] 解决了这个问题，其是基于分布式内存计算框架 Spark[19] 实现的图计算引擎，使得图计算可以融入整个数据流处理的框架.相对于分布式计算框架，GraphChi[20] 则追求在单机完成大规模图数据的计算问题.在图计算领域还有许多研究旨在提高图计算框架在特定问题的处理性能，如文献[21-25]等.学术界对图计算的研究热情普遍高涨.

图数据库专注于图数据的管理，提供高效的图遍历查询(graph traversal).图数据模型则是指图数据库如何以图的方式对数据进行抽象[24]，应用较广泛的图数据模型有 RDF① 模型和属性图模型.RDF 全称为 Resource Description Framework，是由 W3C 制定的知识描述标准，使得按 RDF 表示的不同数据源可以进行数据交换或合并.RDF 中的数据单元是由主语、谓语、宾语组成的三元组，可以直接对应为图上的一条边，因此将 RDF 模型归类为图数据模型.常见的 RDF 存储有 Jena②、AllegroGraph③ 等.当今的图数据库大都采用属性图模型设计[26]，如 DEX[27]、GraphChi-DB[28]、Neo4j、Titan 等，其中 DEX 和 GraphChi-DB 都只是单机数据库，Neo4j 的开源版本也是单机的，只适合处理中等规

① RDF 1.1 Concepts and Abstract Syntax. https://www.w3.org/TR/rdf11-concepts 2017,3,29

② Apache Jena. http://jena.apache.org 2017,3,29

③ AllegroGraph. http://franz.com/agraph/allegrograph 2017,3,29

模的图. Titan 的底层实现了可插拔的存储接口,可部署在 HBase、Cassandra 或 BerkeleyDB 之上,因此可以很好地与 Hadoop 集群结合,组成统一的数据处理框架. 然而,现有的图数据库都没有考虑含有大量重边的属性图应用场景. HybriG 架构填补了这个空白,为含有大量重边的属性图应用场景提供了一种可行的解决方案.

9 总 结

本文提出了一种基于 Titan 和 HBase 的复合存储架构——HyBriG,可以高效处理含有大量重边的属性图. 相比于传统图数据库 Titan 的实现,HyBriG 在处理大量重边时拥有更优异的查询性能,在数据导入方面也有不错的表现.

在获得高性能的同时,所带来的牺牲就是简化了对事务性的支持. 然而,含有大量重边的许多应用场景并不需要很强的事务性支持,HybriG 架构提供了最终一致性,足以满足这些场景的应用需求.

在实践中,明略数据的关联数据分析平台 SCOPA 基于 HybriG 架构开发了核心存储部件. 验证了 HybriG 架构在处理大量重边时的优异性能.

参 考 文 献

[1] Arodriguez M, Neubauer P. Constructions from dots and lines. Bulletin of the American Society for Information Science and Technology, 2010, 36(6): 35-41

[2] Liu Qiao, Li Yang, Duan Hong, et al. Knowledge graph construction techniques. Journal of Computer Research and Development, 2016, 53(3): 582-600(in Chinese)
(刘峤, 李杨, 段宏等. 知识图谱构建技术综述. 计算机研究与发展, 2016, 53(3): 582-600)

[3] Chang F, Dean J, Ghemawat S, et al. Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems, 2008, 26(2): 205-218

[4] Angles R, Gutierrez C. Survey of graph database models. ACM Computing Surveys, 2008, 40(1): 1-39

[5] Lakshman A, Malik P. Cassandra: A decentralized structured storage system. ACM SIGOPS Operating Systems Review, 2010, 44(2): 35-40

[6] Webber J. A programmatic introduction to Neo4j//Proceedings of the Conference on Systems, Programming, and Applications: Software for Humanity. Tucson, USA, 2012, 1(1): 217-218

[7] Peng D, Dabek F. Large-scale incremental processing using distributed transactions and notifications//Proceedings of the Operating Systems Design and Implementation. Vancouver, Canada, 2010: 251-264

[8] Vogels W. Eventually consistent. Communications of the ACM, 2009, 52(1): 40-44

[9] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters//Proceedings of the Operating Systems Design and Implementation. San Francisco, California, USA, 2004: 10-10

[10] Cui Bin, Mei Hong, Ooi B C. Big data: The driver for innovation in databases. National Science Review, 2014, 1(1): 27-30

[11] Page L, Brin S, Motwani R, Winograd T. The PageRank citation ranking: Bringing order to the web. Stanford University InfoLab, Stanford, USA: Technical Report 66, 1998

[12] Leskovec J, Lang K J, Dasgupta A, Mahoney M W. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. Internet Mathematics, 2009, 6(1): 29-123

[13] Chiba N, Nishizeki T. Arboricity and subgraph listing algorithms. SIAM Journal on Computing, 1985, 14(1): 210-223

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

[15] Low Y, Gonzalez J, Kyrola A, et al. GraphLab: A new framework for parallel machine learning//Proceedings of the Uncertainty in Artificial Intelligence. Catalina Island, USA, 2010: 340-349

[16] Gonzalez J, Low Y, Gu H, et al. PowerGraph: Distributed graph-parallel computation on natural graphs//Proceedings of the Operating Systems Design and Implementation. Hollywood, USA, 2012: 17-30

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

[18] Gonzalez J, Xin R, Dave A, et al. GraphX: Graph processing in a distributed dataflow framework//Proceedings of the Operating Systems Design and Implementation. Broomfield, USA, 2014: 599-613

[19] Zaharia M, Chowdhury M, Franklin M J, et al. Spark: Cluster computing with working sets//Proceedings of the Workshop on Hot Topics in Cloud Computing. Boston, USA, 2010: 10-10

[20] Kyrola A, Blelloch G E, Guestrin C, et al. GraphChi: Large-scale graph computation on just a PC//Proceedings of the Operating Systems Design and Implementation. Hollywood, USA, 2012: 31-46

[21] Xu Ning, Chen Lei, Cui Bin. LogGP: A log-based dynamic graph partitioning method. Proceedings of the VLDB Endowment, 2014, 7(14): 1917-1928

[22] Shao Yingxia, Cui Bin, Chen Lei. Parallel subgraph listing in a large-scale graph//Proceedings of the International Conference on Management of Data. Snowbird, USA, 2014: 625-636

[23] Shao Yingxia, Chen Lei, Cui Bin. Efficient cohesive subgraphs detection in parallel//Proceedings of the International Conference on Management of Data. Snowbird, USA, 2014: 613-624

[24] Xu Ning, Cui Bin, Chen Lei. Heterogeneous environment aware streaming graph partitioning. IEEE Transactions on Knowledge and Data Engineering, 2015, 27(6): 1560-1572

[25] Shao Y, Cui B, Ma L, et al. PAGE: A partition aware engine for parallel graph computation. IEEE Transactions on Knowledge and Data Engineering, 2015, 27(2): 518-530

[26] Angles R. A comparison of current graph database models// Proceedings of the International Conference on Data Engineering. Arlington, USA, 2012: 171-177

[27] Martinez-Bazan N, Muntes-Mulero V, Gomez-Villamor S, et al. Dex: High-performance exploration on large graphs for information retrieval//Proceedings of the 16th ACM Conference on Information and Knowledge Management. Lisbon, Portugal, 2007: 573-582

[28] Kyrola A, Guestrin C. GraphChi-DB: Simple design for a scalable graph database system — on just a PC. Computing Research Repository, 2014, abs/1403.0701: 1-12

HUANG Quan-Long, born in 1991, M. S. candidate. His current research interests focus on data warehouse and distributed storage.

HUANG Yan-Xiang, born in 1990, Ph. D. candidate. Her current research interests focus on social media analysis, real-time recommendation.

SHAO Ying-Xia, born in 1988, Ph. D. His current research interests focus on large-scale graph analysis.

MENG Jia, born in 1982, M. S. His current research interests focus on distributed storage.

REN Xin-Qi, born in 1984, M. S. His current research interests focus on big data computing and visualization.

CUI Bin, born in 1975, Ph. D., professor. His current research interests focus on database, social media analysis and large scale graph mining.

FENG Shi-Cong, born in 1973, Ph. D. His current research interests focus on big data analysis and data mining.

Background

The past decades have witnessed the massive growth of Internet. Vast amount of graph data was produced by the boom of social network, e-commerce and online education. Systems like Pregel, GraphLab and PowerGraph are proposed for distributed graph computing. Graph databases like Neo4j and Titan are developed for management of property graph. The property graph is a directed, labeled graph with multi-edges. Vertices and edges can be associated with any number of properties. Since it can represent graph data in most scenarios, the property graph model has numerous applications in industry. However, traditional graph databases encounter significant performance degradation when the graph contains large amount of multi-edges. We reveal the cause of this in Titan, an open source distributed graph database which has attracted wide attention in industry. Titan stores graphs in adjacency list format, where a graph is stored as a collection of vertices with their adjacency lists. Each entry of the adjacency list stores an edge. When querying about adjacent vertices, Titan has to look through the entire adjacency list of the source vertex, which hurts the performance when the multi-edge set grows explosively. For high level queries that are based on adjacent vertices query, e. g. path queries, local cluster coefficient queries etc., the performance loss will be worse.

We propose HybriG, a better distributed architecture based on Titan and HBase for this scenario. HybriG stores the vertex data and graph structure in Titan, the edge data in HBase, respectively. HybriG keeps a concise adjacency list about the graph structure, which helps to gain an order of magnitude improvement in execution of adjacent vertices based queries and batch loading of edge set. In this article, we introduce how HybriG implements its storage layer upon Titan and HBase, and how HybriG transforms graph operations into Titan and HBase APIs. Besides, HybriG achieves high performance for inserting batch of graph data. Furthermore, we introduce the implementation of data consistency between Titan and HBase. Extensive experiments have been conducted to show the outstanding performance of HybriG.

This work is supported by the National Natural Science Foundation of China under Grant No. 61572039, the National Basic Research Program (973 Program) of China under Grant No. 2014CB340405, and the Shenzhen Government Research Project under Grant No. JCYJ20151014093050503 2.