计 算 机 研 究 与 发 展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2016.********
卷(期):起止页,年

分布式大图处理系统中的索引设计与实现

The Design and Implementation of Index on Distributed Big Graph Processing System

Abstract With the rapid growth of technologies like web, social network, and location based service, graph data exist in a variety of applications. There are some distributed graph processing systems that perform pretty impressive when applied to offline graph data analysis tasks which often require complex computation and massive communications between vertices, leading to considerable network traffic. Google Pregel and Apache Giraph are instances of those systems. However, these graph processing systems are mostly based on Bulk Synchronized Parallel Computing(BSP) model, which means that every vertex in the graph is supposed to meet a barrier right after every single iteration during computation, which brings efficiency loss. On the other hand, Pregel-like systems only index vertices by their IDs, which means that when querying properties other than ID, the only way to access vertices that meet some certain requirements is to scan all the vertices. This may result in a high latency which is a severe problem in online querying. Thus, it is our major concern to improve Pregel-like graph processing systems so that we can obtain a better performance on querying any properties on these systems. We investigate on various index mechanisms that fit the graph data querying on distributed systems and apply one of them to the Apache Giraph Framework. The improved system supports both synchronous and asynchronous execution. Finally, we validate the superiority of our proposed mechanism by experimenting on massive synthesized graph data.

Key words distributed graph processing system, big graph, index, non-ID property, synchronization/ asynchronization

摘要 随着Web技术、社交网络、基于位置的服务等互联网技术和应用的发展，大规模图结构数据的存在越来越广泛。一些现有的大规模图数据计算系统能够较高效地完成在离线环境下的大规模图数据分析任务。其中，Google Pregel及其开源实现Apache Giraph已经在实际应用中发挥了极大的作用。这些系统大多基于整体同步并行计算模型，即BSP模型。然而现有系统存在两个主要问题：首先，基于BSP模型的系统在每轮迭代计算间都有同步过程，导致在并行执行效率方面存在提升空间。其次，例如Giraph等类Pregel图数据处理系统对节点的存储是唯一基于节点ID的。对于涉及复杂的条件、非ID属性的查询，只能遍历图中所有的节点，从而导致很大的开销。在图数据的重要性日益凸显的今天，对于图数据的查询的需求也在变得旺盛。尤其是其中复杂的、涉及节点非主属性的查询的高效处理方法的需求变得更加迫切。针对上述问题，我们采用在分布式大图处理系统中增加索引机制的办法来解决。我们首先分析设想了一种分布式构建B+树的增量式构建方法，并根据我们的实验平台Giraph改进了一种更加高效的批量式B+索引的算法。在数据查询阶段，我们在原有消息机制的基础上扩展出收发异步消息的接口，并利用异步消息检索我们的索引。最后，我们基于现有的分布式环境Apache Giraph实现上述索引机制，用实验评估数据分布、索引自身结构对于索引构建阶段的时间代价的影响，并比较我们改进的Giraph系统和原始Giraph系统在查询阶段的时间代价，验证了我们提出的索引机制的有效性。

关键词 分布式；大图；索引；非关键码属性；同步异步
中图法分类号 TP311.13

收稿日期:；修回日期:
基金项目:
通信作者:

随着 Web2.0 时代的到来，互联网上的信息海量增长。论坛、博客、视频网站等社交网站每天都有成千上万的用户贡献丰富的信息。一方面，互联网的用户正在不断增长；另一方面，与用户相关的数据也在快速增加。这些数据很多都可以表示成图结构。在图中，节点可以代表任何实体，节点之间的边代表实体之间的联系，节点的属性集合描述了该节点的特征，从而构成一个属性图网络。例如，在社交网络中，每一个社交网络用户可以作为一个节点，用户之间的好友关系可以作为节点之间的边，而用户所具有的一些属性（如姓名、性别、年龄、毕业院校、现居地等等）作为节点的属性集。属性图作为一种灵活的数据模型，在日常生活、科学技术和商业等各个领域都广泛存在。

基于这样的现状，学术界和工业界产生了多种大规模图数据处理系统。其中技术成熟、应用广泛的系统有 Pregel[1]，Giraph[2]，GraphX[3]，GraphChi[4]，GraphLab[5], GPS[6]等。这些系统对于离线的数据分析任务有着优异的性能。然而，这些系统并不能非常高效地完成在线的、涉及非 ID 属性的复杂查询请求。考虑：在一个社交网络中，筛选出年龄在 14~35 岁之间爱好篮球的男性，在这些用户中投放篮球运动装备的广告。在上面提到的大规模图数据处理系统中，对于这样的查询请求往往需要遍历所有节点，并对相关属性（年龄、爱好）作出判断，以便筛选出符合要求的节点，最后让这些节点执行相关的操作（投放广告）。对于海量图数据来说，上述系统需要遍历所有的节点，并判断该节点的属性是否满足查询中的要求，导致整个查询过程的代价庞大。为了在分布式大图处理系统中实现关系型查询以满足应用需求，迫切需要一套有效的机制以实现分布式大图处理系统在执行复杂的结构化查询时对数据的高效检索。

本文通过为系统建立索引的方式，来解决上述问题。本文主要贡献如下：

1. 负载平衡的分布式大图数据索引的构建方法：提出一种面向大图处理系统的索引设计以支持复杂查询的高效执行，并通过将索引构造任务分担到若干个负责创建索引的根节点，消除了单点承担整个索引构造任务产生的单点性能瓶颈。

2. 面向实时查询的分布式大图数据异步查询方法：通过扩展原有的系统，支持异步消息的发送接收，在查询阶段利用异步接口加速消息的传递，从而加速查询过程。

3. 大规模图数据集上的实验：通过大量在随机人工图上的实验，验证我们的索引机制能够正确地检索出满足条件的所有数据，并且在检索时间上能够比原有的同步遍历方法表现出更高的性能。

1 相关工作

本章介绍与我们工作相关的背景知识、概念，以及与分布式环境中的索引相关的一些工作，包括 Giraph 大图数据处理系统以及已有的分布式环境中的索引工作，并在此基础上阐明现有工作的一些不足之处及我们工作的主要关注点。

1.1 Apache Giraph 系统

Apache Giraph 是一套迭代式大规模图数据处理系统。

Giraph 的计算任务的输入是一个由节点和有向边组成的图（图 1）。例如，此时节点可以看成人（用户），边可以看成加好友请求。节点和边上都关联数值。因此，Giraph 的输入不仅确定了图的拓扑结构，也确定了图中节点和边的初始值。

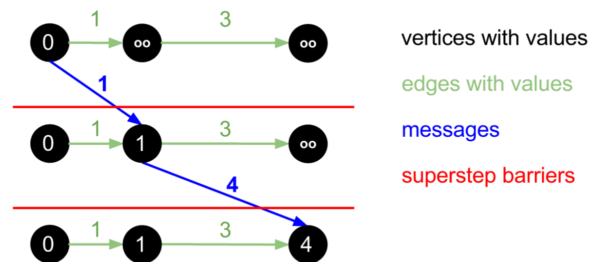

图 1 Giraph 执行任务框架

举例说明，我们需要计算某个确定的人s到社交网络图中任意一个人的距离。在这个计算过程中，边E的值可以是一个浮点数，表示两个相邻的人之间的距离，节点V的值也可以是一个浮点数，表示当前节点代表的人与s之间的距离的上界。源节点s的初始值是 0，其它节点的初始值是无穷大。

Giraph 由于采用了 BSP[7][8][9][10]计算模型，它的计算通过一系列的迭代序列的形式处理。这里的迭代称为“超步”。初始时，每个节点都是“活跃的”。用户面向图中的节点描述其计算任务。每个超步中，每个活跃的节点调用由用户定义的 Compute()方法。Compute()方法应明确三个任务的具体执行指令：（1）接受上一个超步发到当前节点的消息；（2）利用收集到的消息、节点当前的值、出边的值进行计算产生新值；（3）若有需要，向其它节点发送消息。Compute()方法不能直接访问其它节点或它们的出边，节点之间的通信通过发送消息实现。

相邻的超步之间设有同步屏障，它保证了：（1）任意当前超步发送的消息只在下一超步到达目标节点；（2）当且仅当所有节点完成当前超步的计算才开始下一超步。

初始输入的图的结构在计算过程中可能会由于

节点和边的增删产生变化。这些变化由 Vertex 类所提供的方法完成。

节点和边的值在超步之间保持不变，当前超步的初始值是上一超步结束时的值。节点可以在完成自己的计算任务之后变为“不活跃”状态。当所有节点都变为不活跃，并且图中没有消息在传递的时候，整个计算任务结束。

回到上面的例子，上面社交网络中用户距离的问题可以抽象为单源最短路径问题。在图 1 中看到，我们从输入得到一条退化为链状的图，初始化之后开始从节点s计算单源最短路径。距离的上界通过消息的形式传播（蓝色箭头）。每一个超步，当前参与计算的节点收集上一轮邻居节点发给它的消息，然后经过相应操作（更新值）之后，发消息给当前节点的所有邻居节点。消息的值为当前节点当前超步更新后的值加相应边上关联的值。在图 1 的例子中，经过三个超步，所有节点的最短路径已经计算完毕，结束整个任务。

Giraph 这些类 Pregel 的系统对于大规模离线数据分析任务有良好的处理性能[11]，但是仍然存在以下问题：

首先，由于 Giraph 只支持在节点 ID 上的索引功能，对于现实中经常出现的涉及非 ID 属性的查询，则需要遍历图中所有的结点，并逐一作出选择。其次，BSP 计算模型只支持同步计算，而同步计算则使得一部分并行度损失。

1.2 分布式数据库中的索引技术

在传统的关系型数据库之外，由于单机计算能力有限，近年来分布式计算逐渐成为大规模数据处理的基础。在分布式系统之上构建的数据库系统为大规模数据存储和管理提供了底层架构支撑。本节将重点介绍 Apache Hbase[12]分布式数据库系统中索引技术的特点、设计思路并分析它的优势与缺陷。Hbase 是 Google Bigtable[13]的开源实现。

hindex[14]是 Huawei 在 Hbase 上实现的辅助索引解决方案。它将一个表上的任意索引分成若干个部分，并且，由于 Hbase 是碎片化存储的，每一个表会被切分成若干个 splits，存放在不同的 Region Server 上，hindex 的存储和维护也是伴随相应的数据表的，这样就减少了耗时的网络通信，解决了全局索引可能产生的瓶颈问题。此外值得注意的是，在负载平衡的调整过程中，索引表是随着它索引的数据表同时参与平衡器的重新调整的。

hindex 索引表的关键码的构成方式是：“region start key + index name + indexed column(s) value + user table rowkey”.

hindex 的主要架构如下：

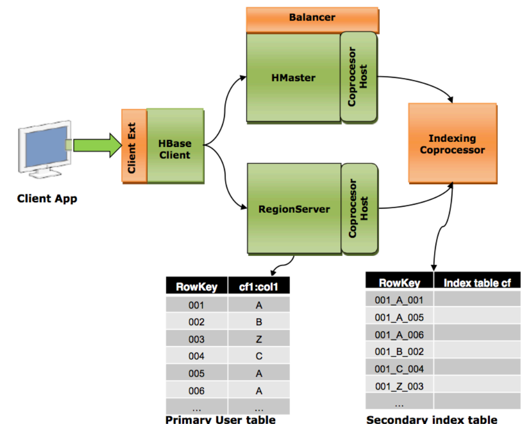

图 2 hindex 的主要框架

索引表关键码可以只包含一列属性，对每一列分别建立，也可以包含多个属性，方便查询涉及多个属性时检索。

hindex 在试验中表现出了良好的性能，对于插入阶段带来的额外代价是常数级别的，并且对于查询阶段所带来的性能提升是很可观的。对于一些条件苛刻的查询，由于返回的条目比较少，性能提升能达到百倍以上。但是由于 Hbase 本身的结构特点，并不适合存储图结构数据。并且 hindex 本质上属于顺序索引结构，随着文件的增大，索引查找性能和数据顺序扫描性能都会下降。面对复杂的图查询，hindex 并不是一个好的选择。

2 分布式大图系统的索引设计和实现

本章主要介绍我们提出分布式大图索引。我们的工作主要基于 Giraph 平台，并采取以节点为中心的编程思想设计索引结构。

我们的设计包括索引数据结构的选取、Giraph 上 B+树索引的增量式和批量式构建，以及通过异步消息对于查询阶段效率的加速。

2.1 数据结构的选取

顺序索引结构[15]及 B+树索引[16]结构是两种常见的索引数据结构。顺序索引结构最大的缺点在于，随着文件的增大，索引查找性能和数据顺序扫描性能都会下降。虽然这种性能下降可以通过对文件进行重新组织来弥补，但是频繁的重组会带来极大的维护代价。B+树索引结构是在数据频繁插入和删除的情况下仍能保持其执行效率的几种使用最广泛的索引结构之一。

B+树节点的内容非常丰富，其占用的存储空间可达到一个磁盘块的大小。树中的每个节点可以有大量指针。因此，B+树一般层数较少而每层节点较多，这就导致在利用 B+树作为索引来加速查询时，读取索

引消耗 I/O 次数相对少，带来的加速是显著的。并且，由于 B+树在结构上也属于图结构的一种，能够非常自然地与被索引的图数据深度融合，嵌入到我们的分布式大图处理系统中，所以我们选择 B+树这种索引效率高，并具有图特征的数据结构来实现我们分布式大图系统中的索引。

我们的 B+树节点实现如下：

表格 1 B+树节点实现

域	含义
isIndexNode	该节点是否索引节点
isLeafNode	该节点是否索引叶节点
value	节点的属性值
fatherNode	父节点指针
ArrayList<Pair> partitionValue	该节点包含的索引码值集合

2.2 分布式 B+树索引概述

在我们的 B+树索引中，用一个标记位来区分索引节点与数据节点。索引树根节点的 ID 固定分配，在由 ID = R 的根节点构建的 B+树分支中，索引节点的 ID 自增式往上加。由于 Giraph 将节点分配到机器的策略采用根据节点 ID 哈希的机制，当 B+树构建完成时，索引节点分布在各个不同的机器上，与数据节点深度融合。我们的索引的这一点特征，使得我们不需要额外考虑索引的逻辑结构设计和物理存储优化，而可以充分利用 Giraph 系统自身对于图数据高度优化的数据结构设计和底层存储。

2.3 BSP 模型同步增量式构建 B+树

由于 Giraph 是一个基于 BSP 模型的图数据处理系统，我们首先尝试在 BSP 同步模型下使用一种增量式的，每次将一个数据项加入索引的算法构建 B+树。它的基本步骤是：

1.每读取一个数据项，找到新的数据项应该插入的叶节点并完成插入。

2.若叶节点所包含的关键码没有达到上限，则算法结束。否则，叶节点分裂成两个节点，并将新生成节点的最大值上传到父节点。若此时父节点所含关键码数量超过上限，则递归执行分裂。否则算法结束。

注意，在 Giraph 的编程框架中，我们需要站在图数据中节点的角度来编写用户定义函数。增量式构建 B+树的过程如算法 1。

```
// 输入：key   数据节点接收的消息，包括要插入的码值、是否分裂阶段
// 输出：无
Function insert(key)

if(key.type() == SPLIT_PHASE)
  // 如当前为分裂阶段，则插入码值，判断是否需要分裂
  this.partitionValue.add(key);
  if(partitionValue.size()>indexDegree)
    newPartition=partitionValue.split();
    addVertexRequest(numNode++, newPartition);
    sendMessage(this.fatherNode, newPartition.max());
else if(key.type() == SEARCH_PHASE)
  // 如果当前是向下查找阶段，搜索正确的子节点转发
  son=findSon(partitionValue, key.value);
  sendMessage(son, key.value);
```

算法 1 BSP 模型同步增量式构建 B+树

这种增量式构建 B+树的算法适合当图中新增一个节点时（例如社交网络中的新用户注册、知识网络中新定义的概念、新上线的网站等）对 B+树索引插入新的索引项的情况，而不适合运行在初始情况下对于大量离线图数据进行索引建立的场景。下面的例子是一个极坏的情况。注意，Giraph 在每个超步之间的同步屏障(barrier)导致了：所有对图结构做出的改变和某一节点在当前超步发出的消息，在下一个超步才会被处理和被目标节点接收。

考虑加入一个点的情况：

1) 假设当前超步为s，需要加入的数据节点发送自身的搜索码值和 ID 到 B+树索引根节点；

2) 第$s+1$超步，B+索引树根节点收到该节点的消息，根据消息中携带的值转发给相应子节点，该步骤持续h超步。其中h为 B+树的高度。

3) 若叶节点不需要分裂，则该索引项插入过程结束，选择下一个数据节点插入。

4) 若叶节点需要分裂，则新建兄弟节点，并上传新建节点的最大搜索码值。值得注意的是，下一超步新建的节点才会加上，并且父节点收到消息，如有必要的话继续分裂、上传新分裂节点的最大搜索。这一过程最坏情况可能持续到根节点，并使 B+树高度增加一层。

那么，每插入一个新的索引项最坏情况需要经过h个超步搜索应该插入的叶节点，$h+1$个超步向上分裂直到根节点分裂，共 $2h+1$个超步。其中h是树高。由于树高在不断变大，代价也会越来越大。并且，每插入一个新的节点，都要阻塞其它节点发送给根节点的消息，因为索引结构在插入该节点的过程中有可能

会发生改变。由于我们是从头开始构建 B+树索引，每个数据节点都要消耗数量如此多的超步，代价是巨大的。在我们的小型实验中，当 B+树度数为 3 时，插入前 10 个节点经过了 24 个超步；前 12 个节点插入需要 32 个超步。当节点逐渐变多时，平均每个节点插入所消耗的超步数逐渐递增，消耗的超步总数增加得越来越快。对于我们在大图系统下的索引构建工作而言，这无疑是低效的。

2.4 改进的 BSP 模型同步批量式构建 B+树

当图数据不需要进行大量增加、删除操作时，增量式构建 B+树索引能够提高数据的查询、修改效率。但是当图数据批量变动，涉及大范围数据的增、删时，B+树索引很可能需要随着数据的变动而进行大量的节点分裂、合并操作，由此带来的大量磁盘读写会导致巨大的索引维护代价，使得上述增量式构建方法非常低效。为解决这个问题，我们进一步提出了一种在 Giraph 平台上运行高效的批量式 B+树加载算法。

批量式加载构建 B+树的主要步骤如下：

1) 新建一个虚拟节点作为创建节点。该节点负责整棵 B+树的创建；

2) 创建节点将所有数据节点按照索引码排序，将排序好的节点的索引码按序写入 B+树索引叶节点中，并保证每一个叶节点中的搜索码尽可能多地被填满，由此形成 B+树的叶节点层。

3) 叶节点层上一层的 B+树内部节点利用叶节点层每个节点中的最大值来构建，并递归构造上层节点直至创建根节点。

4) 当图的结构发生变化，新增少量节点时，使用前文的增量式排序；如果新增的节点数量达到足够多（比如原有节点数量的 20%时），把该图上原有的所有非 ID 属性索引全部删除，重新运行批量加载构建 B+树算法，构建这些非 ID 属性的辅助索引。

```
// 输入：messages    根节点收到的消息
// 输出：建立好的B+树索引
Function BatchBPTreeConstruction(messages)

Sort(messages);
upperMessages = new ArrayList();
while(messages.size() > idxDegree)
    //外层循环是对层数的循环，每循环一次建立一层索引节点
  while(message.size() > idxDegree + idxDegree/2)
    // 当剩余节点较多，则取与B+树度数同样多个节点构成父节点
    newPartition = partitionValue.fetch(idxDegree);
    addVertexRequest(numNodes++, newPartition)
    upperMessages.add(newPartition.max());

  while(message.size() in
        [idxDegree,idxDegree+idxDegree/2])
    // 当剩余节点不够取出idxDegree个节点，则均分成两份
    newPartition1 = partitionValue.fetch(idxDegree);
    addVertexRequest(numNodes++, newPartition1)
    upperMessages.add(newPartition1.max());

    newPartition2 = partitionValue.fetch(idxDegree);
    addVertexRequest(numNodes++, newPartition2)
    upperMessages.add(newPartition2.max());

  // 当前层产生的节点作为下一次循环需要处理的节点
  exchange(messages, upperMessages);
return rootNode of the B+ tree
```

算法 2 改进的 BSP 模型同步批量式构建 B+树

算法 2 中的批量式 B+树构建算法解决了 2.3 增量式构建过程中由于图结构频繁改变和消息量多产生的超步过多问题，但是它产生了一个新的问题：对索引项的排序和整个索引的构建都由单一节点在一个超步之内完成，在这个超步中该节点的计算任务过大，单点消耗时间太长，而其它节点在这个超步中只是空置，而没有任何计算任务。那么在这一个构建超步中负载过于不均衡，会导致构建时间太长。

为了解决这个问题，我们扩展 B+树索引为 B+森林索引。具体来说，我们建立多个根节点。每个根节点负责对固定数量的数据节点创建一棵 B+树。在索引森林的构建过程中，由于每一棵 B+树索引的数据节点是互不重叠的，我们为了保证检索的正确性，也必须保持每棵树中索引节点的 ID 不重叠。为此，我们做了如下的简单计算：

假设N为数据节点总数，D为索引 B+树的扇出，索引森林中T棵树。

每一棵树的叶节点个数为$n = {}^{N}/_{TD}$;

$\Rightarrow$树高为$\log_D n$

$\Rightarrow$每一棵树的节点总数不超过$D^0 + D^1 + D^2 + \cdots + D^{\log_D n}$

$$= {}^{1 - D^{\log_D n+1}}/_{1 - D}$$

$$= {}^{nD - 1}/_{D - 1}$$

$$= {}^{\frac{N}{TD}D - 1}/_{D - 1}$$

$$= {}^{N - T}/_{TD - T}$$

⇒也就是说，在保证索引节点 ID 最小值大于数据节点 ID 最大值的基础上，我们只需要使森林中每棵树根节点的 ID 间隔大于 $N-T/_{TD-T}$ 即可。

最终，我们根据节点 ID 将该节点哈希到某个根节点，由根节点负责它的索引构建。于是，我们保证所有节点 ID 不冲突的情况下，将构建全图 B+森林索引的任务分配到 T 个节点上而非单一节点，有效减少了单点任务过大的问题。

2.5 异步查询阶段处理框架

由于原始 Giraph 系统主要目的是处理数据分析任务，我们需要利用 BSP 模型来保证节点数据的修改不影响程序的正确性。在 BSP 模型的约束下，所有节点必须处于同一个超步，故每个节点必须等待图中所有节点当前超步结束后，才能开始下一个超步。这样就导致了计算任务已经结束的节点的计算资源闲置。然而，由于查询任务对于数据不作更改，我们可以优化系统执行方式，使之在执行查询任务时支持异步执行，任意节点执行完当前超步的任务之后，立马可以进入下一超步，增加系统的并行性。

我们在原有的系统下扩展了发送异步消息和处理异步消息的接口，支持异步消息查询的直接反馈，使得查询阶段同时支持同步和异步。

我们的查询由一个区别于索引节点和数据节点的计算节点 BatchBPTreeIndexMasterCompute 发送到索引森林的每个根节点，并行地完成查询过程。在这个阶段，我们采用异步消息的发送、接收。节点接收异步消息并处理，一旦完成便可发送，不必等待当前超步所有参与计算的节点完成计算任务。由于查询阶段只涉及搜索码的检索和消息的转发，而不设计图中数据的修改，异步消息将提高查询的效率。

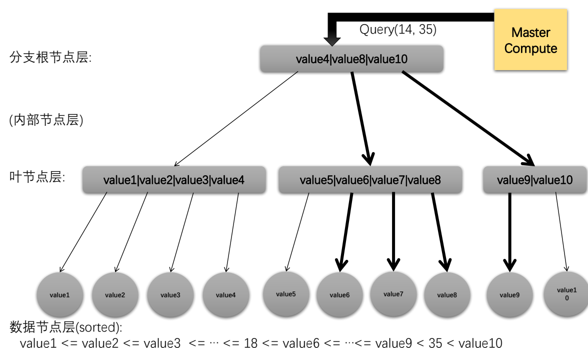

图 3 异步查询执行框架

如图 3，在索引构建阶段，n 棵 B+树同时在一个超步之内完成构建，BatchBPTreeIndexMasterCompute 节点在索引构建完成之后，向索引森林发送查询。

索引节点在接收到异步消息之后，对自身搜索码序列进行检索：若查询类型为范围查询，则找到起始搜索码，向所有可能包含范围内数据项的所有子节点转发消息；点状查询可化归为区间长度为 0 的范围查询。若接收消息的节点为数据节点，则说明该数据节点满足查询要求，完成后续计算任务。

这里我们做了一个优化，当检索范围完毕后，转发消息时，对于发送到非首尾分割码对应的子节点时，可以确定下属所有数据节点都在查询范围内，之后这些子节点的后代（包括这些子节点）可以直接向所有它们的全部子节点直接转发查询消息，不再需要进行范围的检索选取部分子节点转发消息。我们通过在消息中携带一个标记域，表明是否还需要筛选子节点发消息，来实现这一优化。

3 实验与结果

为了考察我们提出的索引机制的构建效率以及与原有 Giraph 系统对比下对于不同类型的查询带来的查询效率提升，我们分别考察了 B+树索引结构对于索引构建时间的影响和 B+树索引对于不同类型的查询带来的查询效率提升。

3.1 实验环境

表格 2 实验环境

节点数量	26
系统	Linux 2.6.32.12-0.7-default
Hadoop 版本	0.20.3
内存大小	48GB
CPU 型号	12CPUs@2.66GHz
硬盘容量	452GB

3.2 实验数据

在所有实验中，我们通过 GTgraph[17]来人工生成图数据。在 GTgraph 的官方开源工具中，我们通过传入命令行参数的方式来控制我们需要生成的图的节点数量、节点平均度数等参数。我们随机生成了节点个数分别为 100、1000、10 万、100 万的随机图，平均度数为 10。

3.3 实验设计与结果分析

3.3.1 构建时间

首先，我们考虑 B+树索引的度数对索引构建时间的影响。我们使用本文提出的方法在 3.2 中的图上构建索引节点度数分别为 5、10、15、20 的分布式 B+树索引，我们在代码中插入检查点来计算得到索引的构建时间。实验结果如下：

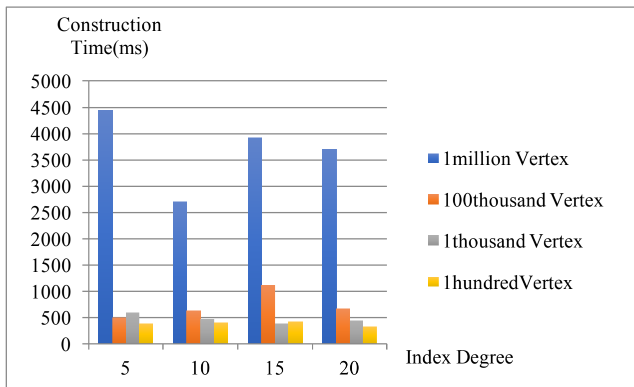

图 4 索引度数对索引构建时间的影响

图 4 中实验数据表明，改变索引的度数，对索引的构建时间观察不到显著影响。由于在索引树数量确定时，索引森林中每一棵索引树索引固定数量的数据节点。当索引节点度数大时，B+树层数少；度数小时，层数多。那么在建立索引的过程中，产生和传递的消息数量也一定。同时可以观察到，构建的时间与图数据中的节点数量成正比。

其次，为了考虑索引中 B+树的数量对构建时间的影响，我们分别在 3.2 中的 4 个图数据集上构建了由 B+树数量为 100、1000、10000、100000、1000000 的索引森林。实验结果如下：

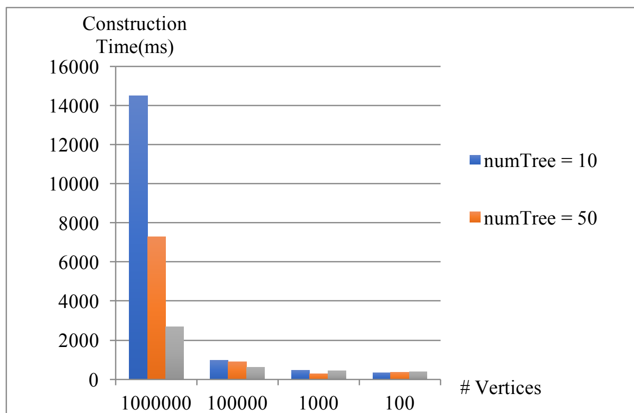

图 5 B+树数量对构建时间的影响

从图 5 中可以看出，索引构建的时间受索引中 B+树的数量的影响比较大。当 B+树数量比较少时，构建时间长；当 B+树数量比较多时，构建比较快。这是由于增加 B+森林中 B+树的数量提高了索引构建阶段的并行度，削弱了单点瓶颈现象。当同时参与构建 B+树的节点越多时，每个节点的计算任务较小，单点瓶颈效应越弱。这说明我们在 2.4 中提出的 B+森林索引方案能够有效加速索引的构建效率。

3.3.2 查询

在评估索引构建的影响因素之后，我们进一步评估索引的结构对于查询耗时的影响。我们首先考虑了索引度数对查询时间的影响，在这组实验中，我们分别设置 B+树节点度数上限为 5、10、15、20，并同样在 3.2 中的 4 个图数据集上实验：

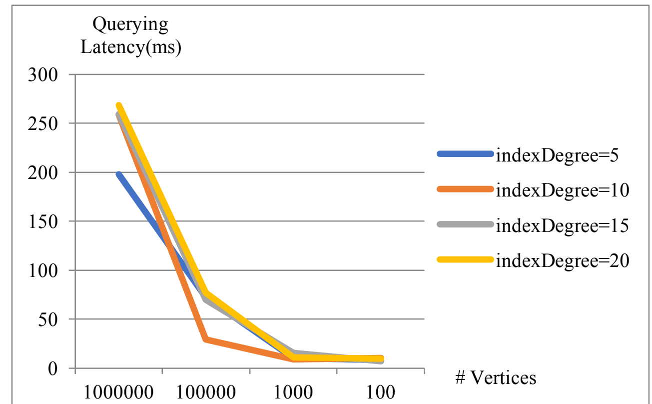

图 6 森林中包含 100 棵树时，查询时间随索引度数变化

从图 6 中可以看出，在 B+树数量确定时，索引的度数对查询时间基本没有影响。当度数变大时，树高会减少，但是每个索引节点检索分割码时时间会增大。

除此之外，我们对比了我们的通过索引的查询和原始 Giraph 同步查询，在单点查询、范围查询的时间代价：

单点查询：

实验数据：节点数为 100 到 1000000 的随机图；随机图中的每个节点有 5 个不同类别的属性，包括字符串、布尔值、整数、浮点数，用以模拟社交网络中用户的姓名、性别、年龄、爱好、标签等属性。

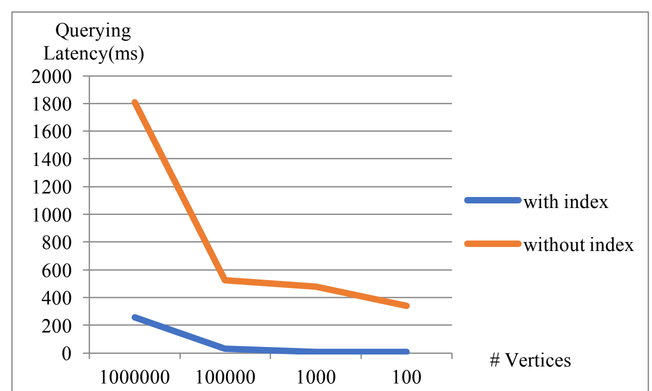

图 7 单点查询同步与异步查询时间对比

从图 7 中可以明显看出，当查询条件为选择出某一个属性为某一确定值的时候，索引异步查询的效率明显高于原始 Giraph 同步查询。

范围查询：我们以查询非 ID 属性$Value \in [18,28]$为例，比较了异步方式与同步遍历方式的查询时间。一下实验在节点总数为 100~1000000 的随机图上完成。实验结果如图 8。

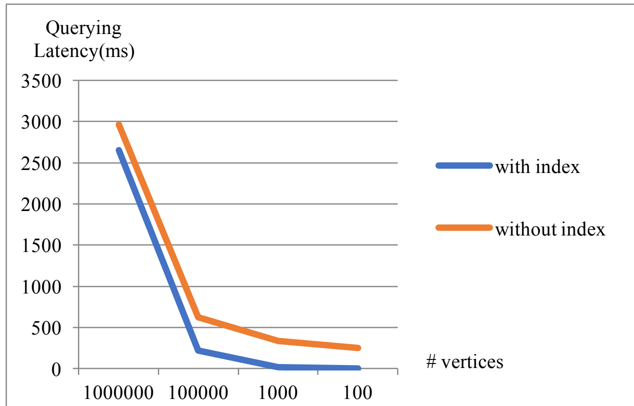

图 8 范围查询同步与异步查询时间对比

结果分析：在范围查询中，异步查询方式效果仍然好于同步方式，但是当图的原始节点数增大，导致处在[18,28]范围内的数据节点增多时，返回给BatchBPTreeIndexMasterCompute节点的消息量成正比增加。由于在异步方式中，节点每收到一条消息就立马开始计算任务，故同时到达的其它消息只能盲目等待，所以异步查询时间产生了无谓损失，与同步遍历方式相比，优势相对减小。

4 总结

本文针对在 Giraph 分布式大图处理系统下，如何高效地处理对非 ID 属性的查询执行进行了分析和研究。首先，我们回顾了主流的大图数据处理系统，选取 Giraph 平台作为我们的实验环境。我们在 Giraph 中支持和引入非 ID 属性的索引，并在数据查询中同时支持同步、异步。在建立索引阶段，我们优化了构建算法，使得构建索引的时间可以按照用户的需求尽可能缩短。在查询阶段，我们针对复杂的查询扩展了现有的系统接口，支持异步消息查询的直接反馈。在检索的时候发送异步消息，使消息传递摆脱了同步屏障的限制，增大了并行度。最后，我们通过在大数据集上的实验证实了我们方法的正确性和高效性。

参 考 文 献

[1] Malewicz, Grzegorz, Matthew H. Austern, Aart JC Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: a system for large-scale graph processing." In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp. 135-146. ACM, 2010.

[2] Welcome to Apache Giraph, [OL]. [2017-05-10] http://giraph.apache.org/

[3] Xin, R.S., Gonzalez, J.E., Franklin, M.J. and Stoica, I., 2013, June. Graphx: A resilient distributed graph system on spark. In First International Workshop on Graph Data Management Experiences and Systems (p. 2). ACM.

[4] Kyrola, A., Blelloch, G.E. and Guestrin, C., 2012, October. GraphChi: Large-Scale Graph Computation on Just a PC. In OSDI (Vol. 12, pp. 31-46).

[5] Low, Y., Gonzalez, J.E., Kyrola, A., Bickson, D., Guestrin, C.E. and Hellerstein, J., 2014. Graphlab: A new framework for parallel machine learning. arXiv preprint arXiv:1408.2041.

[6] Salihoglu, S. and Widom, J., 2013, July. Gps: A graph processing system. In Proceedings of the 25th International Conference on Scientific and Statistical Database Management (p. 22). ACM.

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

[8] Gerbessiotis, A.V. and Valiant, L.G., 1994. Direct bulk-synchronous parallel algorithms. Journal of parallel and distributed computing, 22(2), pp.251-267.

[9] Bisseling, R.H. and McColl, W.F., 2001. Scientific computing on bulk synchronous parallel architectures.

[10] Bulk synchronous parallel, [OL]. [2017-05-28] http://en.wikipedia.org/wiki/Bulk_synchronous_parallel

[11] Facebook, Inc. Facebook Reports First Quarter 2015 Results[R]. California: MENLO PARK, 2014.

[12] Welcome to Apache HBase™, [OL]. [2017-05-10] http://hbase.apache.org/

[13] Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A. and Gruber, R.E., 2008. Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 26(2), p.4.

[14] hindex - Secondary Index for HBase, [OL]. [2017-05-10] https://github.com/Huawei-Hadoop/hindex

[15] Bayer, R. and McCreight, E., 1970, November. Organization and maintenance of large ordered indices. In Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control (pp. 107-141). ACM.

[16] Comer, D., 1979. Ubiquitous B-tree. ACM Computing Surveys (CSUR), 11(2), pp.121-137.

[17] GTgraph: A suite of synthetic random graph generators, [OL]. [2017-05-10] http://www.cse.psu.edu/~kxm85/software/GTgraph/