一种基于最大流的分布式存储系统中查询任务最优分配算法

徐 毅[1] 王建民[1),2),3)] 黄向东[1),2),3)] 董一峰[1] 康 荣[1] 乔嘉林[1]

[1)](清华大学软件学院 北京 100084)

[2)](大数据系统软件国家工程实验室 北京 100084)

[3)](工业大数据系统与应用北京市重点实验室 北京 100084)

摘 要 分布式存储系统多采用数据分区和多副本机制来处理海量数据并提供高可用性.为了提高读写效率,现有系统在将任务分发给不同节点时往往需要考虑数据分区的情况,并使得任务分配能够保证数据本地性.然而,给定一个需要访问多个数据分区的查询任务,现有系统没有充分考虑节点的实际负载情况,导致虽然任务的分配满足数据本地性,但集群查询响应速度仍受到制约.该文提出一种在分布式存储系统中查询任务的节点分配算法,该算法不仅考虑了数据本地性,还利用了多副本机制确保节点间的负载均衡.算法的基本思想是将任务分配问题转化为最大流问题,并通过二分查找寻求最优分配方案.在实验阶段,该文首先通过模拟实验验证该算法的正确性,之后将该算法集成到 Cassandra 中作为一种新的负载均衡策略,并与 Cassandra 原生的两种策略进行性能对比.实验证明,该文提出的算法使得查询性能优于 Cassandra 原生的策略,平均查询时间缩短为原有策略的 50%,某些情况下可以缩短为 11%.

关键词 数据分区;数据本地性;查询优化;最大流;负载均衡;分布式存储系统

中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2019.01858

An Optimal Query Task Assignment Algorithm Based on Maximum Flow for Distributed Storage System

XU Yi[1] WANG Jian-Min[1),2),3)] HUANG Xiang-Dong[1),2),3)] DONG Yi-Feng[1]
KANG Rong[1] QIAO Jia-Lin[1]

[1)](*School of Software, Tsinghua University, Beijing* 100084)

[2)](*National Engineering Laboratory for Big Data Software, Beijing* 100084)

[3)](*Beijing Key Laboratory for Industrial Bigdata System and Application, Beijing* 100084)

Abstract With the continuous development of computer technology, the architecture of data storage system has gradually evolved from centralized to distributed. Most of the distributed storage systems adopt data partitioning and multi-replica mechanisms to deal with large amounts of data and provide high availability. In order to improve the efficiency of reading and writing, existing systems often consider data partitioning when distributing tasks to different nodes and ensure data locality. For a given query task which requires to access multiple data partitions, if reading data on one partition is viewed as an independent subtask, the whole query task can be divided into multiple subtasks. For each subtask, a replica node is selected according to a certain policy. However, policies of existing distributed storage systems do not fully consider actual load of each

收稿日期:2017-08-29;在线出版日期:2018-08-14.本课题得到国家重点研发计划项目(2016YFB1000701)、国家自然科学基金(61802224,U1509213)、北京市科委创新基地培育与发展专项(Z171100002217096)和教育部博士后科学基金(2017M620784)资助.**徐 毅**,硕士研究生,主要研究方向为大数据系统管理.E-mail:xuyi556677@163.com.**王建民**,博士,教授,博士生导师,中国计算机学会(CCF)高级会员,主要研究领域为大数据与知识工程.**黄向东**(通信作者),博士,主要研究方向为大数据系统管理与建模.E-mail:sainthxd@gmail.com.**董一峰**,硕士,主要研究方向为大数据系统管理.**康 荣**,博士研究生,主要研究方向为大数据系统管理.**乔嘉林**,博士研究生,主要研究方向为大数据系统管理.

node when assigning subtasks to different nodes. For example, some allocation strategies may cause many subtasks to select the same node for data reading, resulting in an unbalanced system load. In spite of meeting data locality, query response speed is still constrained. The main reason is that subtask allocation will affect each other because of data partitioning and multi-replica mechanisms. To solve this problem, this paper proposes a node allocation algorithm for query tasks in distributed storage systems, this algorithm not only takes account of data locality, but also takes advantage of multi-replica mechanism to ensure load balance. The basic idea is to transform task assignment problem into maximum flow problem, and find the optimal allocation scheme through the binary search. This paper first considers an ideal situation when all the nodes are idle before a query task is generated. In this case, task assignment problem can be modeled as a flow network. The optimal solution which minimizes query latency is equivalent to find the maximum flow in flow network. This paper gives a mathematical proof. Based on the discussion of ideal situation, this paper has studied a more complex situation when all the nodes have their own load before a query arrives. Under such circumstance, this paper concentrates on how to normalize the load of each node and put them into flow network. Besides, the optimal solution is no longer equal to calculate maximum flow. This paper proposes an algorithm which obtains the optimal solution through continuous iteration. In most cases, the complexity of this algorithm is $O(\log_2(k)\times k^3)$ and k is the number of subtasks. In the section of experiment, this paper first verifies the correctness of the algorithm on simulation data set and compares it with three existing algorithms, then integrates the algorithm into Cassandra as a new load balance strategy and compares it with Cassandra's original two strategies. It shows that the algorithm proposed in this paper performs better than Cassandra native strategy, query time can be shortened to 50% in average. In some cases, it can be reduced to 11%. It can also be observed from experimental results that the algorithm in this paper always assigns new tasks to idle nodes to execute as far as possible.

Keywords　data partitioning; data locality; query optimization; maximum flow; load balance; distributed storage system

1　引　言

随着计算机技术的不断发展，数据存储系统从过去的集中式逐渐向分布式方向发展. 以 HDFS[1]、Cassandra[2]为例，这些主从式或对等式的数据管理系统往往通过采用数据分区[3]和多副本机制[4]为实际应用提供高可靠、高可用、高效的数据读写能力. 此外，这些系统都采用任务分配和结果汇总的方式充分发挥分布式的并行执行能力.

当存在数据分区时，保证“数据本地性”[5]是多数集群进行任务分配时的策略. 数据分区机制是指数据集被划分成多个部分，不同部分存储在不同的节点上. 在实际场景中，用户的查询请求往往需要访问多个分区的数据. 例如，在根据字段 id 进行分区的系统中执行以下查询请求“select id, value from user_table where id in [1,2,3,7,8] and value>10”时，集群需要访问至多 5 个数据分区（某些不同 id 值可能属于同一分区）. 为了减少节点间的数据传输，现有系统往往采用“保证集群中各节点仅处理本地数据（即数据本地性）”这一原则进行任务分配. 例如，若上述 5 个 id 值分别属于节点 $s_1\sim s_5$ 管理，则将上述查询分成 5 个子查询交给 $s_1\sim s_5$ 执行再进行结果汇总.

当数据存在副本时，上述任务分配策略不再唯一：系统在生成查询计划时，为每个分区仅需要选择一个副本节点进行本地数据读取即可. 假设有 k 个分区要读取，每个分区对应 N 个副本，可能存在 N^k 种方案.

然而，给定一个数据分区，选择不同的副本会带来不同的集群查询响应延迟，即某些节点可能同时管理多个分区的副本，并且涉及这些分区的查询请求均被分配至这些节点，从而使得这些节点由于过载成为系统性能瓶颈. 现有的分布式系统（如 Cassandra、

Spark 等)在处理需要访问 k 个分区的查询时,通常把这样的查询看作是 k 个独立子任务.对于每一个子任务会按照一定的策略分配一个副本节点.这些分配策略可能导致许多子任务选择了相同的节点进行数据读取,造成系统负载不均衡.可见,由于数据分区和多副本机制的存在,子任务的分配结果会互相影响,因此在进行任务分配时不仅要考虑数据本地性,还需要考虑多副本机制带来的负载均衡问题.

为了解决上述问题,本文提出了一种基于最大流[6-7]的任务最优分配算法.针对需要访问多个分区的查询,该算法在确保数据本地性分配的前提下,通过分析每个节点自身的负载和数据副本的放置策略,得出查询的节点分配结果,使得集群负载保持均衡,进而消除过载节点,降低查询响应延迟.理论和实践证明本文的算法能够产生一个全局最优的任务分配方案.

本文第 2 节介绍相关工作;第 3 节给出面向数据分区与多副本系统查询优化问题的形式化定义;第 4 节介绍本文需要用到的背景知识、流网络和最大流问题;第 5 节给出理想状况下,即系统无初始负载时的最优分配算法的推导证明;第 6 节给出更接近真实场景的问题建模,即在一个有初始负载的系统中,如何计算最优分配方案;第 7 节将进行实验,首先通过模拟实验验证算法的正确性,之后将算法集成到 Cassandra 系统中,并与原有策略进行对比;第 8 节总结本文的工作,同时展望未来的研究方向.

2 相关工作

由于需要读取多个分区的查询请求十分常见,现有的开源系统都实现了各自的数据分区策略、副本放置策略和任务分配策略.

HDFS 是目前应用最广泛的分布式文件系统,它在存储数据时会先将数据切分成若干数据块,并将不同的数据块放置在不同节点上.每一个数据块对应多个副本,在决定副本的放置策略时(三副本情况下),通常是将第一个副本放到本地机架的一个节点上,第二个副本放到本地机架的另一个节点上,第三个副本放到另外一个机架上去.给定一个需要读取多个数据块的任务,对于每个数据块,HDFS 首先找到它的所有副本,然后采用贪心策略,选择"逻辑距离"最近的那个节点去读取[8].文献[9]认为这样的策略过于简单,在选择副本的时候没有将节点的负载因素考虑进去.针对添加/删除数据块,添加/删除节点这四种操作,该文献对每一种操作引起数据副本的变化提出了一种对应的负载均衡策略,这种策略在选取新的副本时会优先选择那些磁盘使用率最小的节点.文献[10]改进了 HDFS 的副本选择策略.在三个副本的情况下,第三个副本默认会随机选择另外一个机架上的节点.在某些特殊的情况下,如果只能从第三个副本节点访问数据,那么这个时候数据传输会占用较多的时间.因此文献[10]中设计了一种算法,评估了不同机架之间节点相互访问的时间开销,在选择第三个副本的时候尽量选和第一、二副本相互访问时间开销小的节点.文献[11]从另一个角度出发,实际应用中有些数据需要被频繁访问,有些被访问的次数较少.因此可以对过去一段时间内每一块数据访问次数进行统计,针对那些频繁被访问的数据,增加它所对应的副本数量.这样在后续访问的过程中,可以将请求分散给更多的副本节点去执行,降低数据访问的响应延迟.

Spark[12]是较为流行的处理大规模数据的通用计算引擎,它对自身的数据集 RDD[13]进行了数据分区处理.对于计算中要用到的每一个分区,Spark 首先获取分区所在的副本节点列表,然后优先选择本地节点,如果没有本地节点,之后从剩下的节点中随机选一个进行读取.因此其本质是贪心与随机选择策略的组合.

Cassandra 是一个分布式的开源数据库系统,在一个表中,Cassandra 会将数据按行键采用一致性哈希[14]进行分区.在处理需要读取多个分区的查询时,Cassandra 有多种策略允许用户进行选择.其中的一种策略是针对每个要访问的分区,首先获取该分区所有副本所在节点的网络延迟,然后选择延迟最小的节点发送该分区的读取请求.因此其本质也是贪心策略.采用基于贪心策略进行副本选择,容易出现的一个问题就是许多任务都会同时选择某些当前负载较小的节点,造成系统负载不均.为了解决这个问题,文献[15]在 Cassandra 上实现了一种新的策略(文中称之为 C3 算法),首先根据客户端与副本节点的一些统计信息(客户端与副本节点的网络延迟、副本节点当前没有完成的任务数等)对副本节点进行打分,然后看每个副本当前的负载是否达到了上限,选出那些没有达到上限的副本节点.综合以上两个信息选出最终要读取的副本节点.然而 C3 算法有一定的局限性,比如计算副本节点的打分函数中,有一些参数需要人为设置,文中通过多次实验决定了这些参数的取值,但因为没有经过理论证明,

所以不能保证在其他实验环境中可以取得同样好的效果.但是C3算法的思路对于解决本文的问题具有一定的借鉴意义,在本文后面的对比实验中,将对比C3算法和本文提出的算法的效果.

由此可见,现有的系统和研究中对于多个分区数据的读取问题,基本都采用了贪心策略的思想,寻求单分区的局部最优解.然而他们都未考虑的问题是由于数据分区和多副本机制的存在,导致子任务的分配结果会互相影响,从而使得集群中各节点负载不等,进而降低集群的整体响应速度.

3　问题描述

本文要解决的问题是在采用了多副本和数据分区机制的分布式存储系统中,给定一个需要访问不同分区上多个数据的查询操作,如何最优地将该查询任务分配给不同的节点,使得查询的响应延迟最小.这类查询操作十分常见.例如,气象数据会按照经纬度进行等量切分,每个切分的块会有一个编号,在一些分布式存储系统(例如Cassandra)中会依据这个编号进行数据分区,对于一个实际的查询"获取云南省某一天的降水量",需要读取云南省(如图1)所包含的编号(分别对应2,5,8,…,33,实际场景中切分的粒度更加精细)中的数据并按指定的某一天进行过滤,不同编号对应不同的数据分区.由于这些编号往往是不连续的,因此它们对应的数据一般存储在不同的分区上.这时就需要合理地将读取多个分区的任务分配给各个副本节点,使得查询的响应延迟最小.

更形式化地,上述问题可以描述为:假设集群中有M个节点,一个查询Q要访问k个分区上的某些数据,可以看做是k个读取任务,记为$Task_1$,$Task_2$,…,$Task_k$,每个任务请求的数据对应N个副本,它们的对应关系$g(Tasks,Nodes)$如下:

$$Task_1\rightarrow\langle Node_{T_{1-1}},Node_{T_{1-2}},\cdots,Node_{T_{1-N}}\rangle$$
$$Task_2\rightarrow\langle Node_{T_{2-1}},Node_{T_{2-2}},\cdots,Node_{T_{2-N}}\rangle$$
$$\cdots$$
$$Task_k\rightarrow\langle Node_{T_{k-1}},Node_{T_{k-2}},\cdots,Node_{T_{k-N}}\rangle$$

经过任务分配后,每个节点$Node_i$会分配到k_i个任务,它们之间的对应关系$f(Nodes,Tasks)$为

$$Node_1\rightarrow\langle Task_{N_{1-1}},Task_{N_{1-2}},\cdots,Task_{N_{1-k_1}}\rangle$$
$$Node_2\rightarrow\langle Task_{N_{2-1}},Task_{N_{2-2}},\cdots,Task_{N_{2-k_2}}\rangle$$
$$\cdots$$
$$Node_M\rightarrow\langle Task_{N_{M-1}},Task_{N_{M-2}},\cdots,Task_{N_{M-k_M}}\rangle$$

记节点i完成所有的k_i个任务共耗时t_i,由于各个节点并行执行,因此查询Q的响应延迟为$t_{all}=\max\{t_1,t_2,\cdots,t_M\}$.本文目标即是寻找一种任务与节点的分配方案$p=\{k_1,k_2,\cdots,k_M\}(k_1+k_2+\cdots+k_M=k)$,以及相应的$f(Nodes,Tasks)$对应关系,使得$t_{all}$最小.

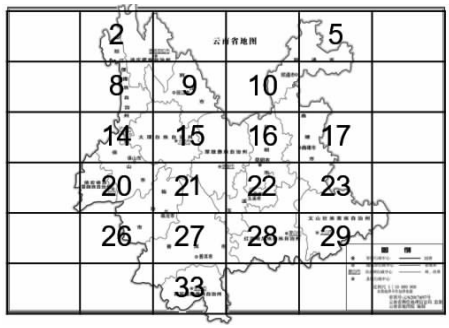


图1　云南省地图

本文提出的这种方法比较适合应用于查询系统,尤其是执行机制是主节点直接向工作节点推送子查询任务的查询系统,例如Impala[16],Greenplum[17],Vertica[18]等这类采用MPP[19]架构的系统.比较适合面对数据多样化且能保证执行时间波动不大的计算任务,例如上面提到的气象数据中的查询过滤任务.

本文仅讨论同构性能节点组成的集群中的优化问题,即在没有其他工作负载的情况下,一个任务在不同副本节点上的执行时间是相同的.在某些场景中,通常在搭建一个集群的时候,机器往往是同一批购买的,配置相同.此外,为便于讨论,本文假定一个查询请求被拆分成k个子任务时,每个子任务的执行时间也相同.例如,HDFS中所有的文件块大小是统一的(如128 MB),很多实际的应用如果要把数据切块进行存储的话,切块的大小也是统一的.因此节点和任务执行时间同构有一定的合理性.其次,实际场景中也肯定存在不同构的情况,比如有些节点的存储介质采用了固态硬盘,读写磁盘的时间会大大降低.但是限于篇幅和时间,本文没有做深入的研究,这部分工作放在未来的工作中进行讨论.

4　背景知识

在图论中,流网络[20]$G=(V,E)$是一个有向图(V是顶点集合,E是边集合),图中的每一条边$(u,v)\in E$有一个非负的容量$c(u,v)\geq 0$,而且如果边集合E包含一条边(u,v),则图中不存在反向边(v,u).如果(u,v)不属于E,那么为了方便起见,定

义 $c(u,v)=0$,并且在图中不允许自循环.定义两类特殊的节点:源结点 s 和汇点 t.假定每个结点都在从源结点到汇点的某条路径上,那么对于每个结点 $v\in V$,流网络都包含一条路径 $s\rightarrow v\rightarrow t$.因此流网络是连通的[21].图 2 是一个流网络的示意图.

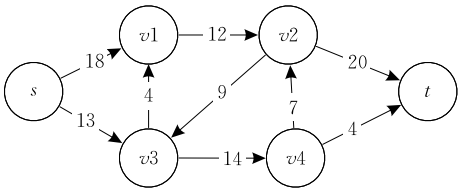


图 2 流网络示意图

设 $G=(V,E)$为一个流网络,其容量函数为 c.设 s 为网络的源结点,t 为汇点.G 中的一个流是一个实值函数 $f:VxV\rightarrow R$,满足以下两条限制:(1)容量守恒,即对于所有的结点 $u,v\in V$,要求 $0\leqq f(u,v)\leqq c(u,v)$;(2)流量守恒,即对于所有结点 $u\in V-\{s,t\}$,要求 $\sum_{v\in V}f(u,v)=\sum_{v\in V}f(v,u)$.当$(u,v)$不属于 E 时,从结点 u 到结点 v 之间没有流,因此 $f(u,v)=0$.流 f 的值就是从源结点流出的总流量减去流入的总流量.在最大流问题[22]中,给定一个流网络 G、一个源结点,一个汇点,希望找到值最大的一个流.现在有许多求解最大流的算法,本文采用的是经典的 Edmonds-Karp 算法[23],这里不再赘述.

5 无负载系统中的最优分配算法

本文首先讨论一种理想情况.假定在开始一个查询任务之前,集群中的各个节点都处于空闲状态,那么一个节点如果被分配到的任务数量越多,它的执行时间就越久.假设有一种分配方案 $p=\{k_1,k_2,\cdots,k_M\}$,其中 k_i 表示第 i 个节点分配到的任务个数,记 $k_{\max}=\max\{k_1,k_2,\cdots,k_M\}$.根据第 3 节的约定,一个节点执行的总时间正比于分配到的任务数,整个集群执行的时间是所有节点执行时间里面的最大值,因此总时间取决于 $k_{\max}$.由此可知,寻找最优分配方案使得查询响应延迟最低,等价于寻找一种任务分配方案,使得 $k_{\max}$ 最小,因此问题就可以转化为寻找最小的 $k_{\max}$.

对于一种分配方案 $p=\{k_1,k_2,\cdots,k_M\}$满足 $k_1+k_2+\cdots+k_M=k$,可以按算法 1 构造出一张流网络图(如图 3).

算法 1. 完全的无负载网流图生成算法.

createNoWorkloadFlowNetworkCompletely(M,k,r,p)

输入:节点数量 M,任务数量 k,任务和副本的对应的关系 r,分配方案 p

输出:完全的无负载网流图 *graph*

```
1.  graph=Array[2+M+k][2+M+k]
2.  FOR(i=0; i<k; i++){
3.    graph[0][i+1]=1
4.  }
5.  FOR (taskID,nodeID) IN r{
6.    graph[1+taskID][1+k+nodeID]=1
7.  }
8.  FOR (i=0; i<M; i++){
9.    graph[1+k+i][1+M+k]=p[i]
10. }
11. RETURN graph
```

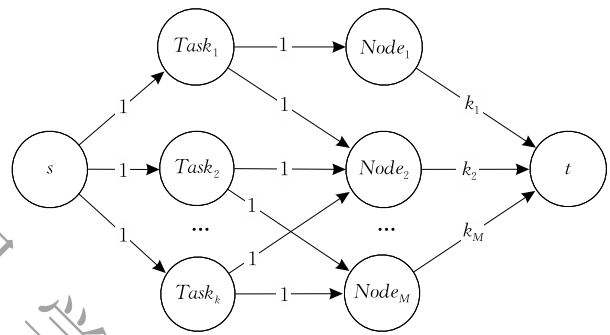


图 3 完全的无负载网流图 G_c

图 3 中包含如下节点:s、t、*Task* 集合和 *Node* 集合.其中源结点 s 表示用户的查询请求;*Task* 集合的数量与查询任务需要被拆分的数量相等(即 k 个),$Task_i$ 代表一个查询子任务;*Node* 集合的数量与集群节点数相同,$Node_j$ 代表集群中的一个节点;汇点 t 代表了最后任务分配的结果.算法 1 在第 2~4 行将 s 到 $Task_i$ 之间连接了一条有向边,它的容量为 1,代表了产生了一个查询子任务.算法 1 在第 5~7 行根据 *Task* 集合和 *Node* 集合之间的边由映射函数 $g(Tasks,Nodes)$,将 $Task_i$ 与若干个 $Node_j$ 之间连接一条有向边,容量为 1,代表了一个子任务和它的副本节点之间的对应关系,该子任务可以分配给其中的一个副本节点去执行.算法 1 在 8~10 行根据分配方案 p,将每个 $Node_j$ 和 t 之间连接一条有向边,容量为集群中节点 j 分配到的任务数 k_j.本文做如下定义.

定义 1. 对于一个分配方案 $p=\{k_1,k_2,\cdots,k_M\}$,

满足 $k_1+k_2+\cdots+k_M=k$，按照算法 1 构造出的一张流网络图称之为完全的无负载网流图，记为 G_c.

在图 3 中，记 $k_{\max}=\max\{k_1,k_2,\cdots,k_M\}$，因为不同节点在本地执行被分配到的子任务时是相互并行的，因此查询的总时间就由 $k_{\max}$ 来决定. 通过对图 3 调用最大流算法，即可得到一个流网络实例来表示 G_c 中的任务流动，如图 4 所示. *Node* 集合与 t 之间的边上增加了新的标记(用括号标记)且与边上的容量相等，表示 *Node* 集合与 t 之间的网络实际流量达到了流网络的最大容量. *Task* 集合与 *Node* 集合之间的加粗边即最终的流向，也即所求的映射函数 $f(Nodes,Tasks)$.

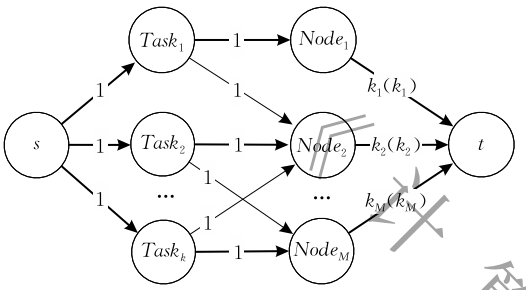


图 4　G_c 的一个流网络实例

然而在实际中，分配方案 $p=\{k_1,k_2,\cdots,k_M\}$ 是未知的，因此无法得到图 4 所示的流网络图. 如果为每个 $Node_j$ 和 t 之间指定一个容量 $c_{\max}$，用于表示 $Node_j$ 可以负责的子任务的最大数量，就可以定义放宽之后的无负载流网络图(如图 5).

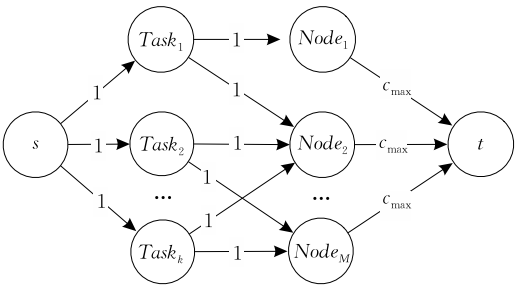


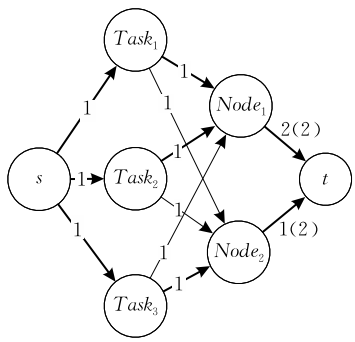
图 5　放宽之后的无负载流网络图 G_w

定义 2.　按算法 1 构造 G_c，并将 $Node_j$ 与 t 之间边的容量改为任意值 $c_{\max}$ 得到图 5，称之为放宽之后的无负载流网络图，记为 G_w.

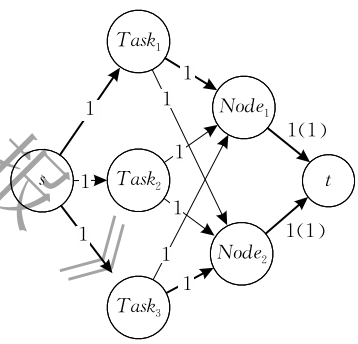
对图 5 调用最大流算法，可得到一个流网络实例，其中 *Node* 集合与 t 之间的真实流量形成了分配方案 $p=\{k_1,k_2,\cdots,k_M\}$. 然而，由于 $c_{\max}$ 是任意指定的，因此 G_w 的最大流可能无法满足用户的查询需求，即 G_w 的最大流小于 k，也即 $k_1+k_2+\cdots+k_M<k$，此时称得到的分配方案 p 是不可行的. 相应地，称 $k_1+k_2+\cdots+k_M=k$ 的分配方案为可行的①.

下面通过一个简单的例子来说明如何通过一张流网络图中的最大流值判断是否存在一种可行的分配方案. 假设现在有三个任务记为 $Task_1$、$Task_2$、$Task_3$，它们都对应两个副本节点 $Node_1$、$Node_2$. 先构造一个 G_w，其中 $c_{\max}$ 待定.

现在令 $c_{\max}$ 等于 2(如图 6(a)，$Node_1$、$Node_2$ 与 t 之间边的容量为 2，并用括号表示)，这时求解这张图的最大流值为 3，与任务总数相等. 图 6(a)中的加粗边表示了最大流为 3 时的一种情况，即 $Task_1$、$Task_2$ 分配给 $Node_1$、$Task_3$ 分配给 $Node_2$. 这样三个任务都被分配到了对应的节点去执行. 因此在最大流等于任务数的时候找到了一种可行的分配方案.



(a) 可行方案



(b) 不可行方案

图 6　最大流求解分配方案的例子

如果令 $c_{\max}$ 等于 1(如图 6(b)，$Node_1$、$Node_2$ 与 t 之间边的容量为 1，并用括号表示)，这时求解得到这张图的最大流值为 2，小于任务数. 通过图 6(b)中的加粗边可以看到，$Task_1$ 分配给 $Node_1$、$Task_3$ 分配给 $Node_2$，但是 $Task_2$ 没有分配给某个节点. 因此在 $c_{\max}$ 等于 1 的情况下无法找到可行的方案.

为了寻找可行的分配方案 p，首先需要确定寻找合理的 $c_{\max}$ 取值. 为此，本文首先给出两个定理.

定理 1.　G_w 中每个节点 $Node_j$ 与汇点 t 之间的最大容量都是 $c_{\max}$，如果 G_w 不存在最大流为 k 的解，

① 根据流网络图原理，因为 S 出发的总流量为 k，因此任意增大 $c_{\max}$，也不会存在 G_w 的最大流大于 k.

那么就不存在一个可行的分配方案 $p=\{k_1,k_2,\cdots,k_M\}$，使得它的最大执行任务数 $k_{\max}\leq c_{\max}$.

证明. 用反证法证明，假设存在一个可行的分配方案 $p=\{k_1,k_2,\cdots,k_M\}$，使得 $k_{\max}\leq c_{\max}$，根据算法 1 可以构造出 G_c 且最大流为 k. 那么对 G_c 放宽一些条件，将每个节点 $Node_j$ 与汇点 t 之间的最大容量提升为 $c_{\max}$，即可得到 G_w，因此 G_c 的解也是 G_w 中最大流为 k 的解. 但是根据题设该 G_w 不存在最大流为 k 的解，因此假设不成立. 证毕.

由定理 1 可知，给定一个 G_w，如果不存在最大流为 k 的解，那么任何一个可行的分配方案的最大执行次数 $k_{\max}$ 都大于 $c_{\max}$. 因此将 $c_{\max}$ 设置过小，使得 G_w 没有可行分配方案是无意义的.

定理 2. 对于一个给定的 $c_{\max}$，如果 G_w 存在最大流为 k 的解，那么就存在一个可行的分配方案 $p=\{k_1,k_2,\cdots,k_M\}$.

证明. 采用最大流算法，可以得到 G_w 图中两两节点间的真实流量. 将每个 $Node_j$ 和 t 之间的最大容量收缩，使得最大容量等于实际流量，即可得到一个定义 1 构造出来的 G_c. 再根据算法 1 的思想就能找到一个可行的分配方案满足要求. 证毕.

本文要求解的是使得最大执行次数 $k_{\max}$ 最小的分配方案，而 $k_{\max}$ 的合理取值是有范围的. 首先每一条边的最大流量不可能超过源节点 s 的发出总数 k，因此 $k_{\max}\leq k$. 由鸽巢原理[24]可知至少有一个节点的分配到的任务数不小于 k/M，因此 $k_{\max}\geq k/M$，所以 $k_{\max}$ 只能在 $[k/M,k]$ 中的整数取值.

基于上述两个定理和 $k_{\max}$ 的取值范围，可以得到求解无负载系统最优任务分配方案的算法.

算法 2. 无负载系统最优任务分配算法.

getNoWorkloadOptimalTaskAssignment(M,k,r)

输入：节点数量 M，任务数量 k，任务和副本的对应的关系 r

输出：任务分配方案 p

```
1. FOR (k'=k/M; k'≤k; k'++){
2.     graph1=createGw(k', M, k, r)
3.     graph2=createGw(k'+1, M, k, r)
4.     IF getMaxFlow(graph1)<k AND
           getMaxFlow(graph2)==k
5.         RETURN getTaskAssigment(graph2)
6.     ENDIF
7. }
8. RETURN getTaskAssigment(createGw(k/M,M,k,r))
```

算法 2 中第 2 行 *createGw*(…)函数表示构建一个定义 2 的 G_w. 第 4 行 *getMaxFlow*(…)函数表示对于一个给定的 G_w，求解它的最大流值. 第 5 行 *getTaskAssigment*(…)函数表示从给定的 G_w 中按照最大流为 k 的解得到一个任务分配方案.

算法 2 的基本思想是从 $[k/M,k]$ 中的整数依次从小到大取，对于每次取到的数 k'，如果以 $c_{\max}=k'$ 构造出来的 G_w 不存在最大流为 k 的解，但是以 $c_{\max}=k'+1$ 构造出来的 G_w 存在最大流为 k 的解，那么最大执行次数 $k_{\max}$ 的最小值就是 $k'+1$，而且满足这种情况的 k' 只会出现一次. 如果不存在这样的 k'，那么最大执行次数 $k_{\max}$ 的最小值就是 k/M.

算法 2 的正确性证明参考附录.

由算法 2 可以确定 $k_{\max}$ 的最小值，且根据这个值构造出来的 G_w 有最大流为 k 的解，再将每个节点 $Node_j$ 与汇点 t 之间的最大容量收缩为实际流量得到一个新的 G_c. 根据算法 1 的思想能找到的分配方案是 $p=\{k_1,k_2,\cdots,k_M\}$ 且 $k_{\max}=c_{\max}$ 是最优的.

6 有负载系统最优化分配算法

在真实系统中，当一个查询 Q 到达时，通常各节点还有一些没有处理完的任务，这些任务可以看作是这些节点的初始负载，第 5 节的算法未考虑这部分负载带来的影响，本节解决这种情况下的分配优化问题.

在有初始负载情况下，每个节点可以附加一个负载系数 L_i，看作是额外的工作量，节点 i 在接到 k_i 个任务后，需要先将 L_i 完成之后再做 k_i，实际执行的任务量就是 k_i+L_i，实际执行时间 $t_i\propto(k_i+L_i)$. 在真实的场景中，任务是不断到来的，每个节点的初始负载 L_i 也是不断变化的，因此在每次开始任务分配之前需要将这个值获取到，然后动态地调整任务分配结果.

此时，在构造类似 G_c 和 G_w 的时候需要做一些调整：将每个节点的负载 L_i 也看作是一个额外的任务加入到流网络中，如图 7 和图 8，其中 $Work_i$ 表示集群中节点 i 初始负载对应的任务，分别称之为完全的和放宽后的有负载流网络图，记为 G_c^* 和 G_w^*. 一种可行的分配方案为 $p=\{k_1+L_1,k_2+L_2+\cdots+k_M+L_M\}$，同时满足以下关系：$k_1+k_2+\cdots+k_M=k$. k_i+L_i 表示集群节点 i 分配到了 k_i 个任务以及它自己要先做 L_i 个任务. 在有负载的情况下，也有类似定理 2 的形式，为方便起见，记 $k+\sum_{i=1}^{M}L_i=Sum$.

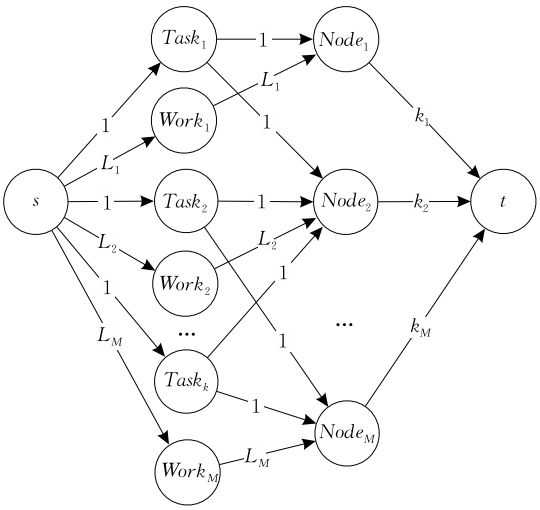


图 7　完全的有负载流网络图 G_c^*

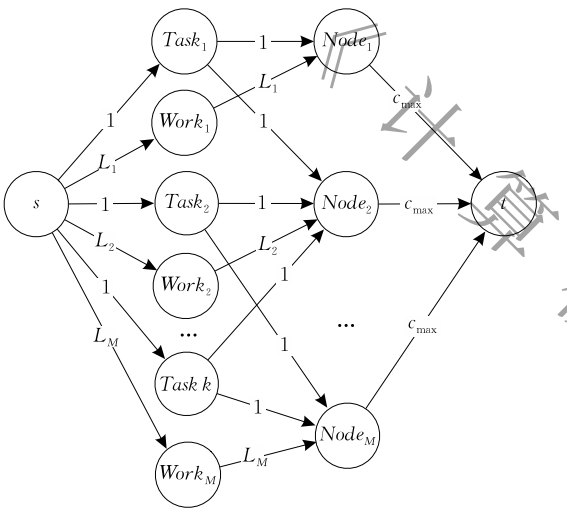


图 8　放宽之后的有负载流网络图 G_w^*

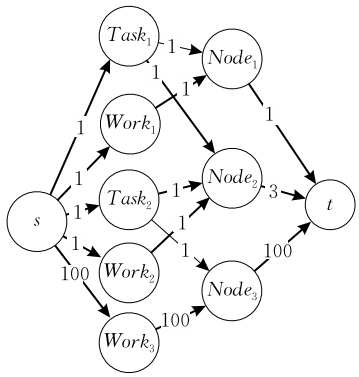
定理 3.　对于 G_w^* 和给定的 $c_{\max}$，如果存在最大流为 Sum 的解，那么就存在一种可行的分配方案 $p=\{k_1+L_1,k_2+L_2,\cdots,k_M+L_M\}$ 使得满足以下公式：$k_{\max}=\max\{k_1+L_1,k_2+L_2,\cdots,k_M+L_M\}\leq c_{\max}$.

记 $\{L_1,L_2,\cdots,L_M\}$ 中的最大值和最小值分别为 $L_{\max}$ 和 $L_{\min}$. 在有负载的情况下，$k_{\max}$ 的最大值不超过 $k+L_{\max}$，最小值不小于 $L_{\min}$. 下面给出有负载的情况下的 $k_{\max}$ 最小值更精确的上界.

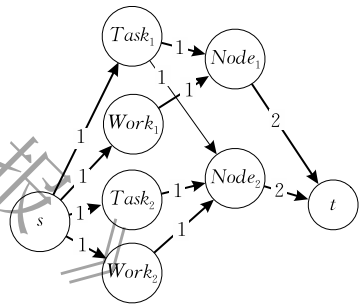
定理 4.　从 $[L_{\min},k+L_{\max}]$ 中依次从小到大遍历整数，对于每次取到的数 k'，如果以 $c_{\max}=k'$ 构造出来的 G_w^* 不存在最大流为 Sum 的解，但是以 $c_{\max}=k'+1$ 构造出来的 G_w^* 存在最大流为 Sum 的解，那么最大执行次数 $k_{\max}$ 的最小值不超过 $k'+1$，而且满足这种情况的 k' 必定存在且只会出现一次.

定理 4 的证明参考附录.

虽然根据定理 4 可以知道 $k_{\max}$ 的最小值不超过 $k'+1$，但是存在图 9(a) 的情况. 因为 $Node_3$ 的负载太大（s 与 $Work_3$ 之间的容量为 100），最大流算法不会分配更多的任务给它，图 9(a) 中 $Node_3$ 只收到了来自 $Work_3$ 的流入，这种情况下根据定理 4 得到 $k_{\max}$ 的最小值≤100，但 100 并不是一个精确的上界；如果将 $Node_3$ 删除（如图 9(b)），就可以得到 $k_{\max}\leq 2$，这时可以发现 $k_{\max}$ 的最小值为 2. 因此当找到一个 $k'+1$ 的时候，根据定理 4 能够找到一个对应的分配方案 $p=\{k_1+L_1,k_2+L_2,\cdots,k_M+L_M\}$，$p$ 中的最大值为 $k'+1$，对所有满足 $k_i+L_i=k'+1$ 的节点，需要检查每个 k_i 是否为 0. 分为两种情况：



(a) 保留节点3



(b) 删除节点3

图 9　有负载流网络图的一个特例

(1) 如果全都不是 0，那么此时 $k'+1$ 就是 $k_{\max}$ 的最小值. 证明该结论需要分两种情况考虑. 第一种情况是：不考虑将图中某些节点删除时，不可能找到比 $k'+1$ 更小的数，满足构造出来的 G_w^* 能够有最大流为 Sum 的解. 因此在不删除节点的情况下 $k'+1$ 就是最小的了. 第二种情况是：考虑删除某些节点能得到更小的值，记为 $k_{\min}$ 且满足 $k_{\min}<k'+1$，设删掉的节点中初始负载的最大值为 $L'_{\max}$，显然 $L'_{\max}<k'+1$，那么用 $c_{\max}=k_{\min}$ 构造出来的 G_w^* 中，将这些删掉的节点补上，并构造出 $c_{\max}=\max\{L'_{\max},k_{\min}\}<k'+1$ 的 G_w^*，依然能够找到最大流为 Sum 的解，和定理 4 就出现了矛盾. 因此如果全不是 0 的情况 $k'+1$ 就是 $k_{\max}$ 的最小值.

(2) 如果至少有一个 k_i 为 0，那么就需要将所有 k_i 为 0 的节点从 G_w^* 中删掉，然后根据定理 4 继续寻找，直到所有 k_i 都不为 0.

本文把上述寻找最优分配方案的方法称作有负载系统最大流算法，可以描述如算法 3 所示.

算法 3. 有负载系统最大流算法.

getWorkloadOptimalTaskAssignment(M,k,r,w)

输入：节点数量 M，任务数量 k，任务和副本的对应的关系 r，节点的负载系数 w

输出：最优分配方案 p

```
1.  WHILE TRUE
2.    k'=search(M,k,r,w)
3.    graph=createMaxflowGraph(k'+1,M,k,r,w)
4.    flag=checkCondition(graph)
5.    IF flag==TRUE
6.      RETURN getTaskAssigment(graph)
7.    ELSE
8.      removeUselessNodes(M,k,r,w)
9.    ENDIF
10. ENDWHILE
```

算法在第 2 行首先用 *search*(…)函数根据定理 4 寻找满足要求的 k'，然后在第 3 行通过 *createMaxflowGraph*(…)函数构造出 G_w^*，第 4 行 *checkCondition*(…)函数会检查是否所有 k_i 都不为 0. 如果是，那么 $k'+1$ 就是 $k_{\max}$ 的最小值，在第 6 行通过 *getTaskAssigment*(…)函数从 G_w^* 中按照最大流为 k 的解得到一个任务分配方案并返回. 如果不是，那么在第 8 行用 *removeUselessNodes*(…)函数删掉所有 k_i 为 0 的节点继续执行循环.

从算法 3 的计算过程可以看出，首先需要知道每个分区对应哪些副本节点，这个时候如果有节点加入或者删除，在这个阶段这部分信息已经获取到了. 然后根据分区对应的副本节点信息构造流网络图，也就是说流网络图是每次动态生成的，而不是预先固定好的，最后求解最优分配方案. 因此，本文提出的算法能够针对集群节点加入或者删除的情况进行自适应，不需要进行额外的操作.

接下来分析算法 3 的时间复杂度. 第 2 行最大流算法采用的是 Edmonds-Karp 算法，时间复杂度是 $O(VE^2)$. 假设节点数为 M，任务数为 k，副本数为 N，则流网络中顶点集合 V 的个数为 $2+2M+k$，边集合 E 的个数为 $3M+k+kN$，则最大流算法的复杂度为 $O((2+2M+k)(3M+k+kN)^2)$，二分查找的次数为 $O(\log_2(k+L_{\max}))$. 如果 $k \gg M$ 的话，算法 3 中一次循环的时间复杂度为 $O(\log_2(k)\times k^3)$. 由于一次循环未必能得到最优解，因此最坏情况下(每次循环都删去一个节点，直到只剩下一个节点)算法 3 的时间复杂度是 $O(\log_2(k)\times k^3\times M)$. 当然，最坏情况一般是不会发生的，正常情况下都是一次就能找到最优解的. 算法的时间开销占查询任务总时间开销的比值将在下面的 Cassandra 实验部分进行评估.

最后要说明的一点是，算法 3 中每个节点的初始负载的计算方法具有一定的技巧性. 可以分成两种情况来考虑.

首先针对某些支持事务的存储系统来说，从一个查询产生到最后执行完，可以看作是一个完整的事务，那么系统应该能够保证在这个查询执行的过程当中不会被其他的操作影响，因此在查询开始执行之前获取到负载的初始值就是当前系统的负载值，而且在该查询执行过程中，这个负载值也不会随之变化. 因此，其他新发出的任务和新完成的任务对系统负载的影响可以不考虑.

其次对于不支持事务的存储系统来说，如果集群中只有一个客户端的话，那么这个客户端看到每个节点的初始负载就是每个节点真正的负载. 但如果同时有多个客户端存在的话，那么每个客户端看到的只是某个时刻节点的负载，因为其他客户端也会有新任务分配或者新任务完成，造成节点实际负载和每个客户端看到的不一样. 所以在多个客户端的场景中，算法 3 中的初始负载是包含了客户端自己看到的节点负载和它根据一些统计信息进行修正后的汇总. 如何根据一些统计信息进行修正有一定的技巧性，在下面的 Cassandra 实验部分将简单介绍本文采用的修正方法.

7 实 验

7.1 模拟实验

本节实现了三种任务分配算法作为基准算法. 第一种是任务随机分配算法，即将任务随机分配给它所对应的副本节点中的一个进行执行(例如 Spark 在本地节点没有数据的时候会从其他副本节点随机选一个读取). 第二种是基于贪心策略的负载最小算法：对于一个任务而言，总是选择当前负载最小的副本节点去执行(例如 HDFS 会认为节点之间的逻辑距离越近，负载越小). 第三种是文献[15]中的 C3 算法，它根据客户端与副本节点的一些统计信息对副本节点进行打分，然后看每个副本当前的负载是否达到了上限，选出那些没有达到上限的副本节点中得分最小的一个. 在 C3 算法的计算过程中，需要人为设定一些参数. 因此在开始下面的对比实验之前，首先为 C3 算法进行了多组参数的测试，

选出了最优的一组.

模拟实验在单机上运行，CPU 型号是 Intel(R) Core(TM) i7-4790K，内存是 8GB. 该实验先设置好模拟实验中集群节点数量 M 和副本数 N，然后为每一个节点产生一个负载，集群所有节点的负载将服从某一种概率分布. 再随机生成一系列任务，每个任务可以分拆为多个子任务，每个子任务对应 N 个副本节点. 实验使用这篇文章提出的最大流算法和其他三种算法进行任务分配，对比集群中分配到任务节点的最大工作负载.

本次实验中集群节点的数量设置为 20，副本数量设置为 3 或 5，这种配置能够模拟出一个小规模应用场景. 模拟的子任务数取值范围是[50,200]，节点初始负载的取值范围是[1,100]. 对于集群节点数为 20 的情况，子任务数假如总数为 100，那么理想状况下平均每个节点分配到的任务数为 5，当某些节点负载较小的时候(比如小于 10)可以认为节点处于相对空闲的状态，那么就可以模拟出最大流算法尽可能多的将子任务分配给这些空闲的节点的情况. 当某些节点负载较大的时候(比如大于 80)，这个时候能够模拟因为某些节点负载过大，算法将不分配任务给这些节点的情况. 因此节点初始负载的取值范围从 1 变化到 100 已经可以模拟出各种情况.

本次实验中一共模拟了两种不同的集群各节点的初始负载分布状况，分别服从均匀分布和正态分布.

7.1.1　均匀分布实验

集群负载服从均匀分布是一种比较常见的状态[25]，从实验的结果(如图 10 和图 11)可以得出几个结论. (1) 随着任务数的增加，每一种算法产生的最大负载都稳定在一个范围之内，而不是随着任务数的增加而递增. 这说明当集群负载服从均匀分布时，每一种算法能够均匀地将任务分配给副本节点；(2) 从四种算法的对比可以发现，最大流算法(实线)、负载最小算法(划线)和 C3 算法(点划线)在副本数为 3 的时候效果相当，在大部分情况下都能同时得到最优解. 只有在极少数的情况下，最大流算法的效果好于负载最小算法和 C3 算法. 因此，在副本数为 3 的情况下，采用负载最小算法是一个比较好的选择，它的计算代价最小；(3) 当副本数为 5 的时候，在任务数较少的情况下(图 11 中任务数小于 130 时)上面的结论仍然成立. 但当任务数比较多的时候，最大流算法的优势就从图中体现出来了. 在任务数比较多的情况下，相比于负载最小算法和 C3 算法，集群的最大负载平均情况下降低了 19%，在最好情况下降低了 39%.

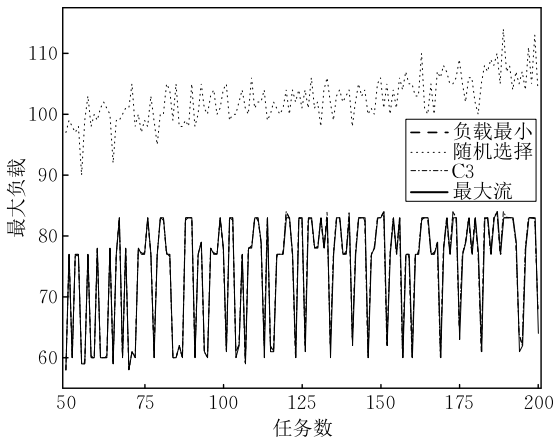


图 10　集群负载处于均匀分布状态，副本数为 3. 图中负载最小曲线(划线)，C3 曲线(点划线)和最大流曲线(实线)基本重合

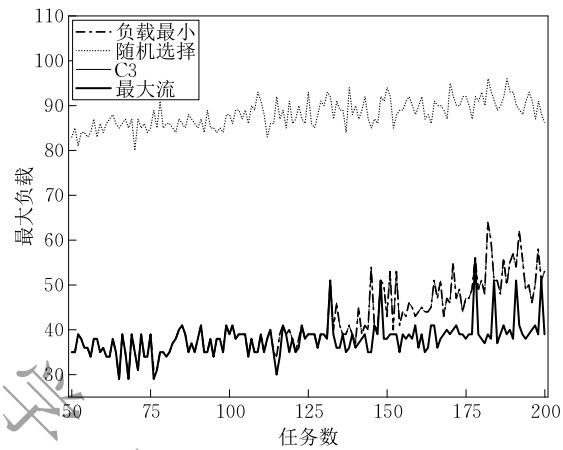


图 11　集群负载处于均匀分布状态，副本数为 5. 图中负载最小曲线(划线)和 C3 曲线(点划线)基本重合

7.1.2　正态分布实验

集群负载服从正态分布是另一种比较常见的情况[26]. 从实验的结果(如图 12 和图 13)可以看到，随着任务数的增加，每一种算法产生的集群最大工作负载都呈现上升的趋势，说明在正态分布情况下，每一种算法不再能够将任务均匀地进行分配. 瓶颈取决于少数几个被分配到任务最多的节点. 在集群负载处于正态分布的情况下，最大流算法的效果要明显好于其它三种算法，当副本数为 3 的时候，集群的最大负载平均情况下降低了 9%，在最好情况下降低了 16%. 当副本数为 5 的时候，集群的最大负载平均情况下降低了 30%，在最好情况下降低了 39%. 还有一点值得注意，在副本数为 5 的情况下，随着任务数的增加，最大流算法的曲线呈现出阶梯状的上升趋势，说明对于新增的一个任务，多数情况下可以在保持最大负载不变的情况下，将该任务分配给一个尚未达到最大工作负载的节点去执行.

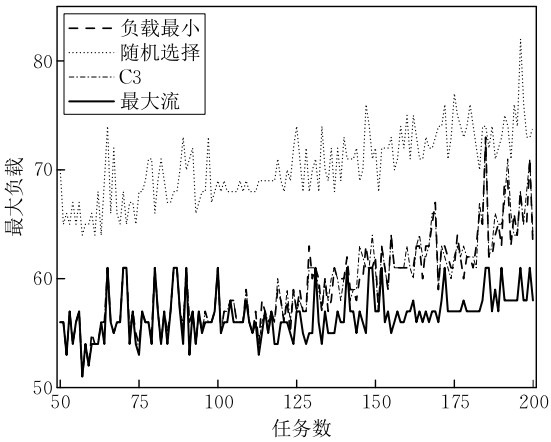


图 12 集群负载处于正态分布状态,副本数为 3

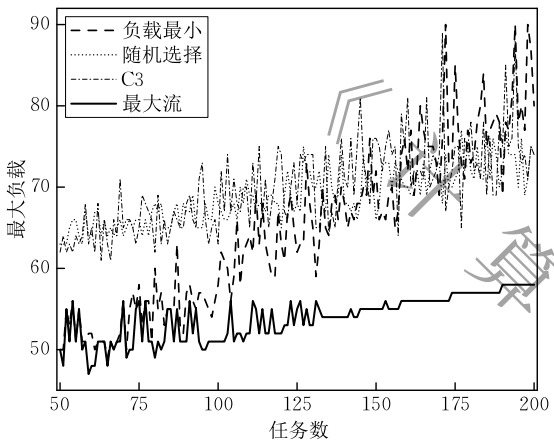


图 13 集群负载处于正态分布状态,副本数为 5

7.1.3 小 结

从上面两个实验的结果可以看到,随着任务数的增多,最大流算法能够有效地降低集群的最大负载.在副本数较多的情况下,最大流算法的表现要好于副本数较少的情况.通过上面章节的理论证明和这里的实验,可以证明最大流算法的有效性.

7.2 真实系统对比实验

本节将最大流算法应用到 Cassandra 系统中,实现一种新的负载均衡策略,称作最大流策略,并与 Cassandra 中原来的 RoundRobin① 策略和 TokenAware② 策略进行对比.之所以选择在 Cassandra 上进行实验,是因为添加一种新的策略时只需要实现相应的编程接口并在查询时指定这种策略,不需要修改 Cassandra 源码.但像在 Impala 这类系统上添加新的策略则需要修改源码,容易造成系统不稳定.

本次实验 Cassandra 的版本是 3.0.13,集群一共有 5 个物理节点,副本数设置为 3,运行时 Cassandra 各项配置参数均采用默认值.每个物理节点的硬件配置相同,一个节点有 8 个 3.60GHz 的 CPU,32GB 内存,5 TB 机械硬盘,网络带宽是千兆.实验过程中在另外三台机器(配置和 Cassandra 集群节点相同)上先使用 YCSB[27] 工具向 Cassandra 中写入十亿条记录,每条记录的大小约为 1 KB.然后再针对这些记录,进行一亿次读写请求,根据读写请求所占的比例不同,模拟 Cassandra 集群处于不同负载状态[28].YCSB 工具产生的读写请求服从 Zipfian 分布,Zipf 参数设置为 0.99.实验中一共进行了三种负载状况下的对比实验,分别对应读频繁(读写比 7∶3)、写频繁(读写比 3∶7)和读写请求比例相同(读写比 5∶5).

查询的任务个数 k 取值范围是[5,50],每一个查询任务需要读取的数据量为 1 MB.实际场景中以气象数据中实时数据自动填图为例[29],针对全国范围内四万多个站点,每个站点有二十多类监测物理量,一小时共有接近一百万个由站点和物理量组成的数值对,需要每隔一段时间(例如一小时)将这些数据写入 Cassandra.直接将这些键值对逐条写入 Cassandra 会严重影响系统的性能,通常的做法是先把这些键值对全部写入到一个文件里面,文件的大小一般是几兆左右,然后再把文件当做一个二进制的字节数组写到 Cassandra 中.这样在后面进行过去一小时全量数据查询的时候,只需要读取一次磁盘.如果要查询某一天的全量数据而数据又是按每个小时来存储的,就可以把这个查询拆分成 24 个子查询来做,然后为每个子查询选择一个合适的副本节点,进而降低总体的查询时间.因此,本文实验中查询任务个数的取值范围和每一个查询任务需要读取数据量的设定值是合理且有一定代表性的.

实验将对比集群在不同的初始负载状态下,对相同的查询采用不同策略所花费的时间.由于最大流策略需要获取集群的负载状态,考虑到实验中的节点 CPU 都是多核的,尽管节点当前有多个任务要处理,但多个任务可以通过调度并行执行,无法直接根据节点待处理任务个数来评估节点的负载状态.因此,本文利用 Cassandra 能够让用户配置的最大并发读写请求个数和通过接口获取当前读写队列长度来归一化每个节点的负载状况,将多个任务在单机上并发执行的模型转化为在单机上顺序执行的模型,解决评估单机负载状况的问题.同时针对第

① RoundRobinPolicy. http://docs.datastax.com/en/developer/java-driver/3.1/manual/load_balancing/#round-robin-policy

② TokenAwarePolicy. http://docs.datastax.com/en/developer/java-driver/3.1/manual/load_balancing/#token-aware-policy

6 节最后分析的多客户端场景(实验中有三台机器作为额外的客户端不断向 Cassandra 发起读写请求),在归一化每个节点的负载时,如果计算得到的节点负载值不是整数,那么就向上取整.表 1 列出了在用 YCSB 工具发送不同比例的读写请求的情况下,Cassandra 集群各个节点的读写队列长度(每个读写比下的左边一列)和归一化后的初始负载值.

表 1　Cassandra 平均读写队列长度和归一化初始负载值

节点编号	读写比 3∶7		读写比 5∶5		读写比 7∶3	
	队长	归一值	队长	归一值	队长	归一值
1	202	7	169	6	184	6
2	104	4	104	4	84	3
3	263	9	182	6	201	7
4	152	5	141	5	196	7
5	90	3	109	4	93	3

在第 6 节的最后评估了最大流算法的时间复杂度,虽然在最坏的情况下将是 $O(\log_2(k)\times k^3\times M)$,但是在本次实验中算法 3 基本一次循环就能找到最优分配结果,所花费的时间都在 10 ms 以内.例如,在图 14、图 15 和图 16 中,当任务数为 15 时,最大流策略计算最优分配方案所花费的时间是 3 ms,但查询时间缩短了 200 ms,查询速度提升了 16%,可以被用户接受.

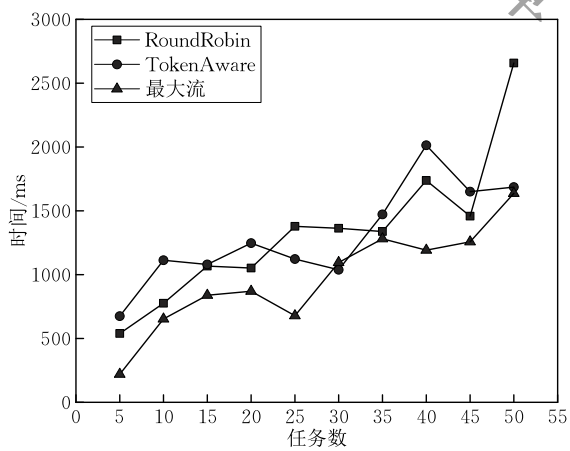


图 14　读写比 3∶7

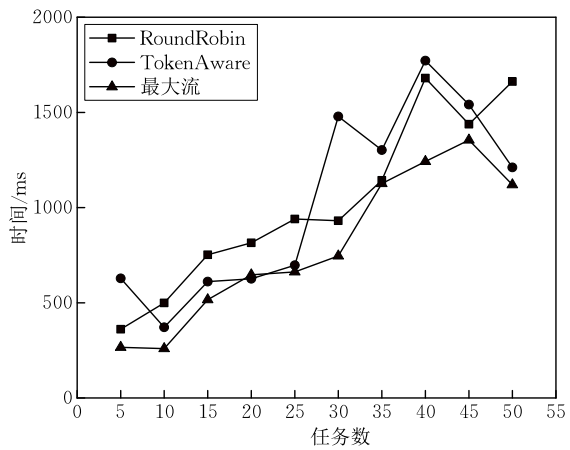


图 15　读写比 5∶5

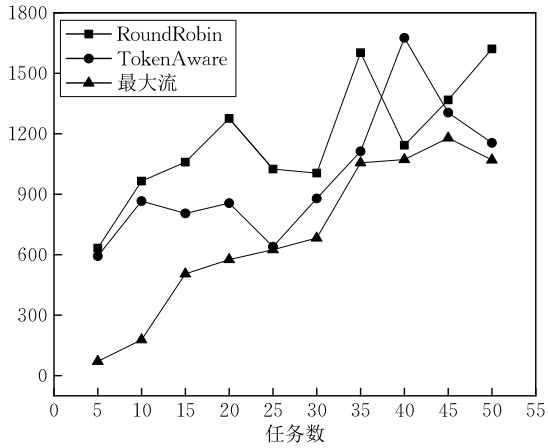


图 16　读写比 7∶3

图 14、图 15 和图 16 列出了不同读写比例下最大流策略和另外两种策略查询时间的对比,从实验的结果可以得出两个结论:(1) 对于给定的某一种负载状况,最大流策略在最好的情况能将查询时间缩短为其他两种策略的 11%,在最差的情况下,最大流策略所花时间和其它两种策略相同.平均情况下,最大流策略能够减少 50%的查询时间.因此,最大流策略的效果明显好于 Cassandra 原来的两种策略;(2) 随着任务数的增加,最大流策略所花费的查询时间呈现阶梯状上升的趋势,和上面模拟实验的结果类似,说明对于新增的查询任务而言,最大流策略能够将其分配给一些比较空闲的节点去执行.

8　总结与展望

本文针对分布式存储系统中查询任务分配问题,提出了一种新的分配算法,在确保数据本地性分配的前提下,通过分析每个节点自身的负载和数据副本的放置策略,得出查询任务分配结果,使得集群的负载保持均衡,进而消除过载节点,降低查询响应延迟.本文先从理论上证明了这一点,之后先通过一系列模拟实验证明了算法的正确性.在 Cassandra 上的实验表明,本文提出的算法的查询性能优于 Cassandra 原生的策略,平均查询时间缩短为原有策略的 50%,某些情况下可以缩短为 11%.对于下一阶段的研究工作,希望能够进一步研究在多客户端场景下中如何准确地归一化节点的初始负载值,准备将网络延迟、节点 CPU 内存资源使用情况等

因素考虑进来,同时还会考虑在集群节点和任务执行时间不同构的情况下,如何评估这些因素的影响.通过在不同真实系统上的实验来说明这篇文章提出的算法具有普适性.

致 谢 感谢审稿专家的宝贵意见!

参 考 文 献

[1] Shvachko K, Kuang H, Radia S, et al. The hadoop distributed file system//Proceedings of the 26th International Conference on Massive Storage Systems and Technologies (MSST). Lake Tahoe Incline Village, USA, 2010: 1-10

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

[3] Ceri S, Negri M, Pelagatti G. Horizontal data partitioning in database design//Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data. Orlando, USA. 1982: 128-136

[4] Wiesmann M, Pedone F, Schiper A, et al. Database replication techniques: A three parameter classification//Proceedings of the Reliable Distributed Systems. Viareggio, Italy, 2000: 206-215

[5] Xie J, Yin S, Ruan X, et al. Improving mapreduce performance through data placement in heterogeneous hadoop clusters//Proceedings of the International Parallel and Distributed Processing Symposium Workshops (IPDPSW). GA, USA, 2010: 1-9

[6] Goldberg A V, Tarjan R E. A new approach to the maximum-flow problem. Journal of the ACM, 1988, 35(4): 921-940

[7] Orlin J B. A faster strongly polynomial minimum cost flow algorithm. Operations Research, 1993, 41(2): 338-350

[8] Borthakur D. The hadoop distributed file system: Architecture and design. Hadoop Project Website, 2007, 11(11): 1-10

[9] Lin C Y, Lin Y C. A load-balancing algorithm for hadoop distributed file system//Proceedings of the 2015 18th International Conference on Network-Based Information Systems (NBiS). Taipei, China, 2015: 173-179

[10] Qin Y, Ai X, Chen L, et al. Data placement strategy in data center distributed storage systems//Proceedings of the International Conference on Computational Science (ICCS). Shenzhen, China, 2016: 1-6

[11] Huo L, Yi R. Research on replica strategy in cloud storage system//Proceedings of the International Conference on Computer Science and Applications. Wuhan, China, 2015: 313-317

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

[13] Zaharia M, Chowdhury M, Das T, 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. San Jose, USA, 2012: 2

[14] Karger D, Lehman E, Leighton T, et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web//Proceedings of the 29th Annual ACM Symposium on Theory of Computing. New York, USA, 1997: 654-663

[15] Suresh P L, Canini M, Schmid S, et al. C3: Cutting tail latency in cloud data stores via adaptive replica selection//Proceedings of the Networked Systems Design and Implementation. Santa Clara, USA, 2015: 513-527

[16] Beine M, Boucher A, Burgoon B, et al. Comparing immigration policies: An overview from the IMPALA database. International Migration Review, 2016, 50(4): 827-863

[17] Waas F M. Beyond conventional data warehousing—massively parallel data processing with Greenplum database//Proceedings of the International Workshop on Business Intelligence for the Real-Time Enterprise. Berlin, Germany, 2008: 89-96

[18] Lamb A, Fuller M, Varadarajan R, et al. The vertica analytic database: C-store 7 years later. Proceedings of the VLDB Endowment, 2012, 5(12): 1790-1801

[19] El-Helw A, Raghavan V, Soliman M A, et al. Optimization of common table expressions in MPP database systems. Proceedings of the Very Large Data Bases Endowment, 2015, 8(12): 1704-1715

[20] Smith D K. Network flows: Theory, algorithms, and applications. Journal of the Operational Research Society, 1994, 45(11): 1340

[21] Cormen T H. Introduction to Algorithms. 3rd Edition. Boston, USA: The MIT Press, 2009

[22] Elias P, Feinstein A, Shannon C. A note on the maximum flow through a network. IRE Transactions on Information Theory, 1956, 2(4): 117-119

[23] Edmonds J, Karp R M. Theoretical improvements in algorithmic efficiency for network flow problems. Journal of the ACM, 1972, 19(2): 248-264

[24] Veldman W, Bezem M. Ramsey's theorem and the pigeonhole principle in intuitionistic mathematics. Journal of the London Mathematical Society, 1993, 2(2): 193-211

[25] Abouzeid A, Bajda-Pawlikowski K, Abadi D, et al. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. Proceedings of the Very Large Data Bases Endowment, 2009, 2(1): 922-933

[26] Moore J, Chase J, Farkas K, et al. Data center workload monitoring, analysis, and emulation//Proceedings of the 8th Workshop on Computer Architecture Evaluation Using Commercial Workloads. San Francisco, USA, 2005: 1-8

[27] Cooper B F, Silberstein A, Tam E, et al. Benchmarking cloud serving systems with YCSB//Proceedings of the ACM Symposium on Cloud Computing. Indianapolis, USA, 2010: 143-154

[28] Rosselli M, Niemann R, Ivanov T, et al. Benchmarking the availability and fault tolerance of cassandra//Proceedings of the Workshop on Big Data Benchmarks. New Delhi, India, 2015: 87-95

[29] Signorelli S, Kohl T. Regional ground surface temperature mapping from meteorological data. Global and Planetary Change, 2004, 40(3): 267-284

附　录.

算法 2.　正确性证明.

证明.　首先考虑 $k'=k/M$ 的情况，这个时候如果构造出来的 G_w 存在最大流为 k 的解，那么根据定理 2 就可以找到一种可行的分配方案，而且最大执行次数为 k/M，因为任何一种分配方案的最大执行次数都是在 $[k/M,k]$ 中取值的，而 k/M 已经是最小的了，在这种情况下，最大执行次数的最小值就是 k/M.

如果 $k'=k/M$ 时，如果构造出来的 G_w 不存在最大流为 k 的解，依次取 $k/M+1, k/M+2, \cdots$，假设 q 是第一次满足 q 不存在最大流为 k 的解而 $q+1$ 存在最大流为 k 的解的整数，由定理 1 可知，因为 q 不满足存在最大流为 k 的解，所以最大执行次数大于 q，由于 $q+1$ 存在最大流为 k 的解，这个时候 G_w 中的节点 $Node_j$ 与汇点 E 之间如果不存在一条边，它的流量等于最大通量为 $q+1$，那边的最大通量收缩为 q，这时候原来最大流为 k 的解仍然成立，与前面的矛盾，因此这样的边必定存在，也就确定了最大执行次数不超过 $q+1$. 综合起来可以确定最大操作次数就为 $q+1$.

下面证明这样的 q 只出现一次，采用反证法. 如果存在一个 r 使得 $q+1<r\leq k$，r 是不存在最大流为 k 的解而 $r+1$ 存在最大流为 k 的解的整数，既然 r 对应的 G_w 不存在最大流为 k 的解，但是 $q+1$ 存在，对于由 $q+1$ 构造出来的 G_w，将节点 $Node_j$ 与汇点 E 之间的最大通量扩容为 r，原来的解仍然成立，因此就导出了矛盾，这样的 r 不存在，则 q 只会出现一次. 该证明的目的是为了在寻找这样 k' 时候可以采用二分查找进行效率的优化.　　证毕.

定理 4.　从 $[L_{\min}, k+L_{\max}]$ 中依次从小到大遍历整数，对于每次取到的数 k'，如果以 $c_{\max}=k'$ 构造出来的 G_w^* 不存在最大流为 $k+\sum_{i=1}^{M}L_i$ 的解，但是以 $c_{\max}=k'+1$ 构造出来的 G_w^* 存在最大流为 $k+\sum_{i=1}^{M}L_i$ 的解，那么最大执行次数 $k_{\max}$ 的最小值不超过 $k'+1$，而且满足这种情况的 k' 必定存在且只会出现一次.

证明.　先记 $k+\sum_{i=1}^{M}L_i=Sum$.

首先证明这样的 k' 一定存在. 当 $k'=L_{\min}$ 的时候，这个时候 G_w^* 汇点的流入容量的最大值只有 $ML_{\min}<Sum$，这种情况下最大流为 Sum 的问题肯定无解，当 $k'=k+L_{\max}$，任何一个节点的流入量都小于 $k+L_{\max}$，因此最大流问题必定有解. 所以一定能够找到一个 k' 满足要求.

如果能够找到满足要求的 k'，因为以 $c_{\max}=k'+1$ 构造出来的 G_w^* 有最大流为 Sum 的解，而且从节点到汇点至少有一条边的流量达到了 $k'+1$，否则由 $c_{\max}=k'$ 构造出来的 G_w^* 有最大流也有解. 那么由定理 3 可以知道，存在一种对应的分配方案，使它的最大操作数 $k_{\max}$ 不超过 $k'+1$.

最后证明 k' 的唯一性. 采用反证法，假设存在另一个整数 r 使得 $k'+1<r<k+L_{\max}$，有 $c_{\max}=r$ 构造出来的 G_w^* 不存在最大流为 Sum 的解，但是以 $c_{\max}=r+1$ 构造出来的 G_w^* 存在最大流为 Sum 的解，那么因为 $k'+1$ 存在解，如果将 $k'+1$ 放宽到 r 的话原来的解也成立，因此有了矛盾. 所以 k' 是唯一的.　　证毕.

XU Yi, M. S. candidate. His research interests focus on big data system management.

WANG Jian-Min, Ph. D., professor, Ph. D. supervisor. His research interests include big data and knowledge engineering.

HUANG Xiang-Dong, Ph. D. His research interests include big data system management and modeling.

DONG Yi-Feng, M. S. His research interests focus on big data system management.

KANG Rong, Ph. D. candidate. His research interests focus on big data system management.

QIAO Jia-Lin, Ph. D. candidate. His research interests focus on big data system management.

Background

With the rapid development of computer technology, the architecture of data storage system has gradually evolved from centralized to distributed. Take HDFS, Cassandra for example, these data management systems often provide reliability, high availability and efficient data access through data partitioning and multi-replication mechanisms. Besides,

in order to improve the efficiency of reading, existing systems often consider data partitioning and replication when distributing tasks to different nodes and ensure data locality.

This paper mainly focuses on optimizing task allocation in query process. For a given query task which requires to access multiple data partitions, if reading data on one partition is viewed as an independent subtask, the whole query task can be divided into multiple subtasks. For each subtask, a replica node is selected according to a certain policy. Existing systems usually randomly pick some nodes to execute rather than fully considering the actual load of each node. In worst case, it may cause many subtasks to select the same node for data reading, resulting in an unbalanced system load. In spite of meeting data locality, query response speed is still constrained. The main reason is that subtask allocation will affect each other because of data partitioning and multi-replica mechanisms.

This paper proposes a node allocation algorithm for query tasks in distributed storage systems, this algorithm not only takes account of data locality, but also takes advantage of multi-replica mechanism to ensure load balance. The basic idea is to transform task assignment problem into maximum flow problem, and find the optimal allocation scheme through the binary search. In most cases, the complexity of this algorithm is $O(\log_2(k)\times k^3)$ and k is the number of subtasks. In the section of experiment, this paper first verifies the correctness of the algorithm on simulation data set and compares it with three existing algorithms, then integrates the algorithm into Cassandra as a new load balance strategy and compares it with Cassandra's original two strategies. It shows that the algorithm proposed in this paper performs better than Cassandra native strategy, query time can be shortened to 50% in average. In some cases, it can be reduced to 11%. It can also be observed from experimental results that the algorithm in this paper assigns new tasks to the idle nodes to execute as far as possible.

This research was supported by the National Key Research and Development Program of China (2016YFB1000701), the Major Program of National Natural Science Foundation of China (61802224, U1509213), Key Projects of Beijing Municipal Science and Technology Commission (Z171100002217096), and the China Postdoctoral Science Foundation (2017M620784).