软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2017,28(3):563−578 [doi: 10.13328/j.cnki.jos.005168]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

分布式数据流上的高性能分发策略*

房俊华[1,2], 王晓桐[1,2], 张 蓉[1,2], 周傲英[1,2]

[1](华东师范大学 计算机科学与软件工程学院,上海 200062)

[2](上海市高可信计算重点实验室(华东师范大学),上海 200062)

通讯作者: 张蓉, Email: rzhang@sei.ecnu.edu.cn

摘 要: 随着大数据应用的普及,高效可扩展的数据流操作在实时分析处理中扮演着越来越重要的角色.分布式并行处理架构是应对大流量、低延时数据流处理任务的一种有效解决方案.然而在 Key-based 分组并行处理中,由于数据的倾斜分布及数据流本身的实时、动态和数据规模不可预知等特性,使得数据流分布并行处理系统存在持续且动态负载不均衡现象,这会造成系统时效性降低、硬件资源浪费等问题.现有的研究工作处理均衡负载有两种方案:(1) 基于 key 粒度的迁移,使得并行处理节点负载达到均衡;(2) 基于元组粒度级别的拆分,采用随机分发使系统均衡.前者将系统调整至给定的均衡容忍范围内,类似于一维装箱的 NP 问题;后者对 key 的拆分势必带来新的为维护 Key-based 操作的正确性而增加的额外代价,如内存及网络通信成本.综合两种方法,提出对 key 按需拆分、尽量合并的方法,通过轻量级均衡调整算法以及保证 Key-based 操作特性的拆分方法,使系统既能达到后者的均衡,又能减少细粒度均衡所带来的额外代价.

关键词: 分布式数据流;负载倾斜;基于 Key 操作;均衡调整;负载迁移

中图法分类号: TP311

中文引用格式: 房俊华,王晓桐,张蓉,周傲英.分布式数据流上的高性能分发策略.软件学报,2017,28(3):563−578. http://www.jos.org.cn/1000-9825/5168.htm

英文引用格式: Fang JH, Wang XT, Zhang R, Zhou AY. High-Performance data distribution algorithm on distributed stream systems. Ruan Jian Xue Bao/Journal of Software, 2017,28(3):563−578 (in Chinese). http://www.jos.org.cn/1000-9825/5168.htm

High-Performance Data Distribution Algorithm on Distributed Stream Systems

FANG Jun-Hua[1,2], WANG Xiao-Tong[1,2], ZHANG Rong[1,2], ZHOU Ao-Ying[1,2]

[1](School of Computer Science and Software Engineering, East China Normal University, Shanghai 200062, China)

[2](Shanghai Key Laboratory of Trustworthy Computing (East China Normal University), Shanghai 200062, China)

Abstract: Along with the popularization of big data applications, scalable and efficient stream join processing plays a more important role in online real-time analysis. The distributed parallel processing framework provides an effective solution which facilitates processing of massive data stream with low latency. For Key-based calculations, data skewness and inherent features of stream data, such as real-time, dynamics and unpredictability on data volume, lead to load imbalance to distributed processing systems. Such phenomenon can produce poor performance and waste hardware resources. There have been two solutions to load imbalance: 1) Key-based migration scheme that keeps balance among parallel processing nodes; 2) tuple-based partitioning scheme that distributes data randomly to achieve load balance. The former scheme adjusts system to the defined equilibrium range, which resembles the one-dimensional packing problem. And the latter maintains the accuracy of Key-based operations, which certainly incurs additional memory cost and network communication cost. This

* 基金项目: 国家高技术研究发展计划(863)(2015AA015307); 国家自然科学基金(61232002, 61332006 ,61572194)

Foundation item: National High-Tech R&D Program of China (863) (2015AA015307); National Science Foundation of China (61232002, 61332006, 61572194)

收稿时间: 2016-07-31; 修改时间: 2016-09-14; 采用时间: 2016-11-01; jos 在线出版时间: 2016-11-29

CNKI 网络优先出版: 2016-11-29 13:35:10, http://www.cnki.net/kcms/detail/11.2560.TP.20161129.1335.010.html

paper presents a novel parallel processing scheme that combines both Key-based and tuple-based schemes to partition keys on demand. The proposed scheme adopts a lightweight load balance algorithm and a partitioning scheme which retains the characteristics of Key-based operations, thus realizing the load balance of tuple-base strategy while reducing the additional cost of fine-grained balance.

Key words: distributed data stream; workload skew; Key-based operation; workload balance; workload migration

当今,随着人们对隐藏在大数据背后价值的重视,使得数据分析的实时性也变得越来越重要[1,2].以微博为例,迅速捕获并分析用户对于热点话题的传播与观点,对舆论的正确引导、谣言的有效制止以及广告的精准投放都会产生巨大价值.然而,传统的离线数据批处理与数据分析工具无法提供数据流业务所需要的实时性能,而流数据处理系统正是应对此类问题的有效解决方案.当前,流数据系统广泛应用于众多领域,例如网络监控管理、金融交易管理、通信数据管理等.随着无线、传感网络以及各种新型应用的发展,数据量急速膨胀,原有的单机硬件配置方案已经不能满足海量数据处理要求.因此,基于分布式环境的并行流处理方案得到广泛关注和研究[3−7].

在分布式流数据处理系统中,由于流数据的动态性、不可预知性以及持续不断性[8],使得数据倾斜导致的负载不均成为抑制系统性能的常见问题之一.数据倾斜是指由于数据在某些属性维度上分布不均,导致当数据依据该维度进行数据切分时出现严重的分块不均匀现象.数据倾斜是并行处理架构中的常见问题,在并行数据库[9,10]以及 MapReduce 并行计算框架中[11,12]均存在因 key 粒度的差异导致节点负载不均的现象.而在分布式流系统中,由于数据流本身具有无界性及动态变化性特征,数据倾斜的情况会随着流系统的运行不断变化,使得分布式流系统中负载均衡问题变得更为复杂.也正是由于这种负载倾斜的持续动态变化,现有基于 MapReduce 框架的单次调整负载均衡策略无法直接应用于并行流处理系统.

在分布式流处理系统中,负载均衡问题主要体现为操作级负载均衡[13−15]和数据级负载均衡[16−20].

- 操作级负载均衡问题出现在早期以操作为负载分发单位的流处理平台中,由于不同操作的时间复杂度不同,较差的执行计划会导致多个复杂操作集中在单一节点,从而出现部分节点过载情况;
- 数据级负载均衡问题则会出现在以数据为负载分发单位的流处理平台[21]中,当操作的执行依赖于数据状态信息时,由于数据在某些维度上分布不均,导致在某些负载分发策略下出现节点数据负载量的差异,即:在相同的操作流程下,出现某些节点数据量大而过载的情况.

当前,海量数据流需要能够实现单个操作横向扩充的大吞吐量处理能力,现有工作也主要集中在数据流量发生变化时如何通过横向扩展节点来降低节点负载.对于这类问题,现有的解决方案分为以下两种方式.

(1) 以 key 为单位粒度进行均衡调整.这种方案进行节点负载均衡时需要迁移整个 key 的状态,典型代表 Readj[17].该方案能够很好地保持 Key-based 操作的语义,但同时,使系统达到均衡状态的灵活性受限;

(2) 以元组为单位粒度进行均衡调整.该方案将并行节点组织成自定义的架构后,将元组随机地分发到各并行节点.随机分发策略从根本上解决了系统因 key 粒度不同而导致的负载不均现象,但其破坏了 Key-based 操作的语义,从而带来了操作的局限性.比如在文献[22]中,它将并行处理节点组织成为矩阵模型,这种架构不存在负载不均衡现象,但它比较适合做连接操作,尤其是 theta 连接.此外,二分图架构[19]将预连接的两条流分开存储,同时将分开存储的两端划分成多个子组,使元组流入子组内是随机分发的.这在一定程度上缓解了负载不均衡现象,但对子组内元组进行操作时,则需要广播的分发方式.PKG[20]将一个 key 拆分为多份后,有选择性地分发至不同并行节点来避免负载倾斜问题.然而,这种拆分无法支持连接操作.

综上所述,在分布式架构的数据流操作中,使基于 key 操作的分布式并行处理节点达到均衡状态正面临着以下难点.

1) 以 key 为单位粒度的均衡调整能够最大限度地保持基于 key 操作的语义,但调整单位粒度大,对系统的均衡性不够友好;同时,当单个 key 的负载大于单个节点的负载时,拆分 key 将是必然的操作;
2) 以元组为单位粒度进行均衡调整有利于系统的负载均衡,但其在一定程度上破坏了基于 key 操作的语义,带来的后果是,这种方案将为维护基于 key 操作的正确性而付出额外的代价,例如矩阵架构[22]由于需要存在复本而浪费内存及网络资源.

本文针对分布式流处理系统下的数据级负载均衡问题展开研究,提出了一种基于数据流数据的动态负载均衡策略,以解决流系统中由数据倾斜现象而引起的并行处理节点负载不均衡造成的问题.主要设计思想是尽量保持基于 Key 操作的语义,这体现在:1) 为使系统达到均衡,选择性地拆分 key;2) 均衡过程中,在不影响系统均衡的前提下,尽量合并已经被拆分的 key.本文的主要贡献归纳如下.

(1) 提出了一种负载均衡调整的架构及相应算法,在此基础上定义了均衡框架中的功能模块,进而确保系统处理的正确性.通过实时监控节点负载状况,在保证尽可能少的数据迁移基础上确保负载的均衡,并通过在数据分发端添加路由表的方式,保证实时到达数据的正确分发,从而实现动态的负载调整;

(2) 通过研究数据倾斜现象对并行处理节点的影响,将影响系统性能的衡量归纳为路由表代价、迁移计划的制定速度、迁移代价和后续网络代价.文中给出了实现单一优化目标和整体优化目标的实现方法,并通过实验验证方法的有效性;

(3) 提出了一种可恢复的 key 拆分方案,该方案最大程度地保持基于 key 操作的语义,进而减少系统为维护操作语义而付出的额外代价;

(4) 通过丰富的对比实验,验证框架正确性和算法的可行性.

本文第 1 节描述背景知识及问题的定义.第 2 节介绍相关的工作.第 3 节详细叙述提出的负载均衡调整的架构.第 4 节介绍具体的迁移计划制定方法.第 5 节通过实验验证文中提出方法的有效性.第 6 节进行总结.

1 相关工作

针对数据倾斜现象导致的负载不均问题,Flux[17]架构提供了一套自适应的解决办法.Flux 借鉴并行数据库的技术,通过添加缓冲机制来应对上游操作发送数据时出现的短期不均衡现象;面对长期不均衡现象,Flux 通过控制器收集各个节点上的闲置率,并进行数据迁移以实现负载均衡.然而,对于如何确保数据分发策略的一致性以及分发算法的优化,Flux 并未给出具体的说明,这在一定程度上影响了策略的完整性.近期,随着分布式流处理系统的成熟,针对流数据系统负载不均衡现象开展了进一步的研究.Readj[18]提出了根据 key 为粒度进行均衡调整的方案:每次均衡迁移时只考虑调整当前负载较大的key,小频率的key全部迁回至哈希路由节点.当系统不均衡时,Readj 根据负载两两配对节点,采用暴力方法对节点内的 key 试着交换或者直接移动来寻求更为均衡的状态.该方法导致的问题是迁移计划制定的时间较长,尤其在对于节点均衡度要求较高以及各个节点负载难以均衡的情况时.PKG[20]将一个 key 拆分为更小的粒度后,有选择性地分发负载至不同并行节点上,从而避免负载倾斜问题.但是这种拆分无法支持依赖整个key的历史状态进行操作的计算,如Join.文献[23]在此基础上进一步探讨了对 key 的拆分策略问题.基于矩阵模型的方法[22,24−26]把并行处理节点组织成矩阵的形式,将待连接的两条数据流 R 流和 S 流分别从矩阵的水平和垂直方向流入,进而确保一条流内的任意元组能与另一条流的所有元组碰面.二分图架构方法[22]将并行处理节点分为两组,分别用于存储 R 流和 S 流;同时,将每一组处理节点分为若干子组来缓解节点负载的倾斜.在基于矩阵模型的架构中,数据在流入时被随机分发至矩阵处理节点的一整行或一整列,尽管系统基本是绝对均衡状态,但该架构中数据的大量复制导致资源浪费严重.基于二分图模型的方法在子组内采用随机的方式分发存储数据,进行连接操作时使用广播的分发方式路由待连接的流数据.在该架构中,数据在子组内的随机分发方式虽有利于并行处理节点的均衡,然而另一条待连接数据流在子组内的广播路由则使该架构的网络通信代价上升.

2 并行数据流处理框架

本节介绍并行数据流的处理架构,描述数据倾斜在架构中引起并行节点负载不均的情况,进而描述设计均衡调整策略应遵守的准则.

2.1 分布式数据流拓扑

数据流(stream)是时间上无界的元组(tuple)序列.基于数据分发的并行流处理系统是以数据流为源头,依据

设定的操作并行度将数据分发到各并行节点处理,并最终获得处理结果的过程.虽然操作的并行化提高了节点资源的利用率,但与此同时,由于数据流的动态性和分布不确定性使数据分发策略可能导致节点间负载不均,从而会造成系统延迟过大的问题[27].

图 1 展示了基于数据分发的分布式流处理系统架构模型,其中,数据路由模块处在每个操作的结束阶段,该模块负责将数据依据分发策略分发至对应节点来执行下一个操作动作.常见的数据分发策略包括随机分发、哈希函数分发、分段分发等.在对 key 进行聚集操作的计算中,随机分发策略无法保证一致性;而当流数据存在数据倾斜现象时,哈希函数分发及分段分发策略不能保证负载均衡.

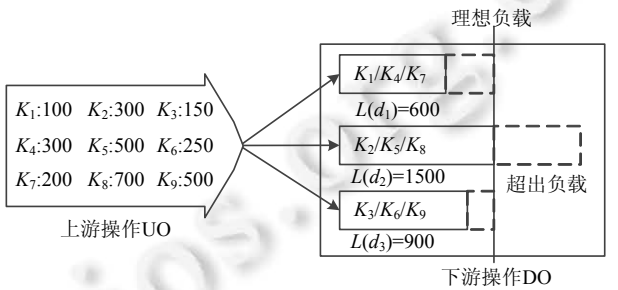


Fig.1 Data skew in partition

图 1 数据分发倾斜

在实际应用中,数据倾斜现象是普遍存在的,如数据在某些属性上存在取值集中或是分段集中的现象,即,在某些值或某段取值所对应的数据非常多.图 1 中:假设数据源在 t 时刻上游操作 UO 发出 k_1~k_9 的元组个数(即 key 的粒度)分别为 100,300,150,300,500,250,200,700,500 条,这些流数据在负载分发操作(假设存在某个哈希函数)中被分发至下游操作节点 DO 上的 3 个并行节点,即 d_1~d_3.理想情况下,所有数据被平均地分配到 3 个负载节点中,从而实现每个节点上的实际负载量都与平均负载相同,即,每个节点上的负载值均为 1 000.然而如图 1 所示,由于按照当前哈希函数的分发方式导致 d_2 接收到远多于另两个节点的数据量,因此出现 d_2 超载的情况.这种状况下,d_2 节点在该操作中的执行时间将远多于另外两个节点.假设下一阶段的操作是对 d_1~d_3 的数据做汇总,那么两个负载较低节点需要等待 d_2 的结束,从而造成 d_1 和 d_3 运算资源浪费,即,系统整体的执行效率受限于 d_2 的执行.

2.2 本文目标

通常,在设计高可用的分布式流处理系统的数据分发策略时,需遵循以下规则.

(1) 负载均衡性.鉴于各并行节点执行的操作相同,节点数据负载将直接决定运算负载,因此,数据分发策略需要具有理论上均匀分配数据的能力.

(2) 分发规则一致性.由于数据流操作中存在如排序、聚合、连接等有状态的算子,而流系统本身具有无界性的特点,因此需要保证在流系统运行过程中,具有相同 key 的元组始终流向符合已定义的语义操作节点,进而确保数据处理结果的正确性.若涉及数据迁移操作,则需要将已有(状态)数据迁移的同时修改数据分发策略,以确保策略一致性.

(3) 算法高效性.分发策略要求具有较低的时间复杂度.数据分发模块存在于系统的各层操作中,且数据流中所有元组在发送过程中都需要进行路由计算处理,因此,算法复杂度高将加大流系统的响应延迟;同时,调整控制模块快速地制定均衡策略对提高系统性能也是非常重要的.

(4) 架构通用性.负载均衡架构对映射、聚合及连接等基本操作必须是通用的,以增强系统的可用性.具体来说,一个应用不可能只有一种操作.由于在这些基本操作中,如连接操作,尤其是 theta 连接较为复杂,通用方案设计困难.

本文旨在设计一套通用的轻量级迁移策略,达到在尽可能减少系统额外开销的情况下保证流系统均衡及

操作正确性的目的.本文后续使用的符号定义见表 1.

Table 1 List of notations

表 1 符号表

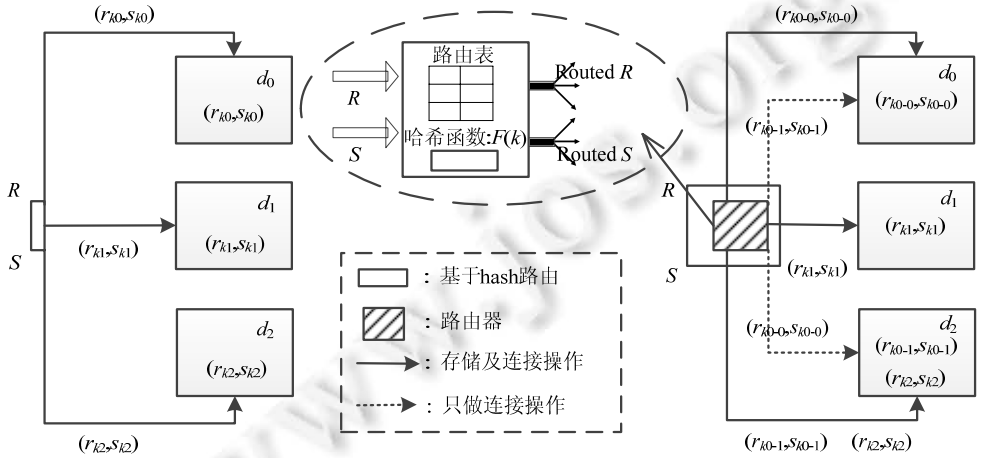
名称	描述	名称	描述
k	元组的 key	τ	负载不均衡容忍度
d	下游操作中的节点	p	谓词
UO	上游操作	Y	路由表
DO	下游操作	$F(\cdot)$	基础路由函数
N_U/N_D	上游/下游操作并行度	$Y(\cdot)$	经路由表映射的节点
$\mathcal{D}$	下游操作并行节点集合	$\bar{w}$	元组
$L(\odot)$	$\odot$的负载	K	key 的个数
$\bar{L}$	平均负载	$K(\cdot)$	节点 d 的 key 的集合
UL	负载不均衡容忍值	MP	迁移计划

3 架构设计

本节描述在本文中设计的数据分发策略,同时给出相关术语的定义及操作代价的分析.

3.1 分发策略

按照给定流数据分发函数(如哈希方法),并行处理节点很可能出现负载不均现象.此时,数据分发的基本思路是对超载节点流量进行切分,将部分超载流量划分至低负载节点.图 2 展示了 3 个并行节点 d_0~d_2 处理 R 流和 S 流中 3 个 key 等值连接操作示例.图 2(a)按照哈希路由的方式将分别属于 R 流和 S 流的 3 个 key 分发到 d_0~d_2 节点上.然而,当第 1 个分支(r_{k0},s_{k0})具有较大的负载而第 3 个分支(r_{k2},s_{k2})具有较小流量时,d_0 和 d_2 便分别成为超载节点和低载节点.此时,均衡调整策略如图 2(b)所示,即:根据实际超载情况,将较大负载(r_{k0},s_{k0})拆分为两部分:($r_{k0\text{-}0}$,$s_{k0\text{-}0}$)和($r_{k0\text{-}1}$,$s_{k0\text{-}1}$),再将它们分别存储于 d_0 和 d_2 节点.此后的路由操作中,($r_{k0\text{-}0}$,$s_{k0\text{-}0}$)和($r_{k0\text{-}1}$,$s_{k0\text{-}1}$)在节点 d_0 和 d_2 中将被区别对待,即:($r_{k0\text{-}0}$,$s_{k0\text{-}0}$)在节点 d_0 中既做连接也做存储操作;而($r_{k0\text{-}1}$,$s_{k0\text{-}1}$)在节点 d_0 中只做连接操作,之后便被丢弃.同理,d_2 节点上进行着与之相反的操作.



(a) 基础路由 (b) 按需拆分 key 模型

Fig.2 Partition-Keys-on-Demand model

图 2 按需拆分 key 模型

为保证操作的正确性,拆分的 key 如何路由至关重要.在本文的架构设计中,数据的路由分发由路由器模块实现.在图 2(b)中,路由器分为基础路由和路由表路由两类.其中,基础路由是一个对 key 的函数映射,诸如散列哈

希、一致性哈希或按区域划分等.路由表记录了在均衡调整过程中移动过的 key 与其被调整到的目的节点间的映射关系,路由表内的记录形如$(k,\{(d_i,q)\})$,即:k 被映射存储到目标节点 d_i 的概率为 q,q 是该节点中存在该 key 的数据量占整个 key 数据量的比值.当数据流流入负载节点前,首先需根据路由表来判断其目的节点的位置:若路由表内存在该元组的映射记录,则其流向由路由表确定;否则,根据基础路由来映射自己的目的节点.需要强调的是:只要一个 key 相对于其基础路由的目的节点发生了改变,不管是整体移动还是拆分移动,那么其所有的信息需添加至路由表.

路由算法见算法 1,路由器按功能划分为两个模块:存储路由模块和操作路由模块.存储路由模块如算法 1 中第 2 行~第 5 行,流入元组首先根据其 key 信息判断路由表中是否存在它的路由信息:若存在,则根据路由表路由将元组存储在该 key 映射的负载节点中;否则,根据基础路由函数来寻找其存储的目标节点.如算法 1 第 4 行、第 5 行,此时,存储路由只返回一个节点作为元组存储的目的地.操作路由模块的功能是根据用户定义的操作语义,在路由时结合操作谓词 p 进行路由选择.如算法 1 第 6 行~第 14 行,若是非等值连接,则元组的路由需同时根据路由表和基础路由函数选择符合操作谓词 p 的目标节点.等值连接时,操作路由类似于存储路由,不同的是,其将元组路由至映射集合中所有的节点.

算法 1. 路由方法.

Input:路由表 Y,基础路由范数 $F(\cdot)$,流数据$\{v\}$,操作谓词 p;

Output:目的结点 d. /*←:赋值;→:装载;$F(k,p)$:根据谓词 p 对 k 的函数映射;$Y(k,p)$:根据谓词 p 在路由表中对 k 的路由;M:对 k 的路由目的节点的集合;$ONE(Y(k,=))$:返回 k 的映射集合当中的一个负载节点 d;$ALL(Y(k,p))$:返回 k 的整个映射集合*/

```
1  k←Extract(w̄)   //将元组的操作 key 提取
2  function STORAGEROUTING(Y,F(·),k)   /*处理存储路由*/
3    if k∈Y then
4      return ONE(Y(k,=))   //将元组存储至路由指向的目标负载节点
5    return F(k,=)   //将元组存储至基础路由指向的目标负载节点
6  function OPERATORROUTING(Y,F(·),k)   /*处理操作路由*/
7    if p≠ then
8      ALL(Y(k,p))→C   //将元组按照操作谓词 p 发送至路由表指向的所有目标负载节点
9      F(k,p)→M   //将元组按照操作谓词 p 发送至基础路由指向的目标负载节点
10     return M
11   else if then
12     if k∈Y then
13       return ALL(Y(k,=))   //将元组发送至所有的路由指向的目标负载节点操作
14     return F(k,=)   //将元组发送至基础路由指向的目标负载节点操作
```

3.2 相关定义

在并行处理系统中,按硬件资源可分为网络、CPU 及内存负载的均衡问题.类似于文献[17],本文认为较多的数据量应该引起系统更多的注意,即,更大粒度的 key 将同时增大对网络、CPU 及内存的消耗.所以,本文从数据量的角度出发考虑系统的负载均衡,即,本文旨在将并行负载节点承载的数据量调整至均衡.对于负载节点 d 的不均衡度τ的定义如下:

$$\tau=\frac{|L(d)-\bar{L}|}{\bar{L}},$$

其中,$L(d)$是节点 d 的负载,$\bar{L}$ 是并行节点的平均负载.在均衡调整过程中,系统较为关心是否有节点的负载超出 UL,因此,系统设定了阈值,即,最大不均衡度$\tau_{\max}$.原因是超载节点会拖慢系统的整体运行效率.因此,系统通常限定并行节点中的最大不均衡度$\tau_{\max}$,即,在并行节点中不能存在比最大不均衡负载 UL 更大的节点负载,UL 可表

示为 $UL=(1+\tau_{\max})\cdot\bar{L}$.本文将操作中并行处理的单元称为节点,例如,storm 中的一个 task 即为本文所述的一个节点.

为了方便本文后续对系统均衡调整的描述,现将在迁移过程中出现的动作定义如下.

- 本节点:对于元组的 key 来说,假设使用基础路由函数将该 key 映射得到的目地节点为 d,则 d 称为该元组或该元组的 key 的本节点;
- 回迁:将处于非本节点的 key 迁移到其本节点的动作称为回迁;
- 移迁:将处于非本节点的 key 迁移到另一个非本节点的动作称为移迁;
- 迁出:将处于本节点的 key 迁移到非其本节点的动作称为迁出.

上述的 3 个迁移动作中,回迁动作会减少路由表的记录数;移迁对路由表记录数没有影响,但需在路由表中更新对应 key 的路由信息;迁出动作需在路由表中添加迁出 key 的路由信息,进而使路由表变大.

均衡调整时,我们将迁移动作对 key 状态的影响结果分为如下 3 种.

- 拆分:迁移动作只移动一个 key 的部分元组;
- 合并:迁移动作使已拆分的 key 又聚集到同一个负载节点上;
- 整体:迁移时将 key 作为一个整体进行移动.

3.3 代价分析

一个高可用的调整策略需要能够在保证负载均衡的同时使系统代价小.对负载均衡调整的代价衡量,可按照均衡调整的时间段,将代价分为调整动作前的决策代价、调整过程中的迁移代价和调整后的运行代价 3 个部分:调整动作前的决策代价即为系统计算迁移计划速度;调整过程的代价是系统根据迁移计划进行调整时付出的迁移代价;调整后的运行代价包括均衡调整后系统维护路由表的代价以及为保证操作的正确性而广播连接请求到拆分 key 的存储节点所付出的网络代价.具体分析如下.

- O_1:迁移计划的制定速度.启动调整模块时,系统则已经处于吞吐量降低的不均衡状态.因此,均衡调整方案是否能够快速制定影响着系统的性能;
- O_2:路由表代价.路由表的存储代价方面,为确保系统的正确性,上游操作 UO 的所有的 N_U 个并行节点都需要缓存路由表;计算代价方面,较大的路由表无疑加剧了元组的路由代价;
- O_3:迁移代价.同迁移计划的制定速度一样,迁移的过程中,系统已经处在影响性能的不均衡状态,此时,轻量级的迁移动作对于携带大量运算状态的操作尤为重要;
- O_4:后续网络代价.较少分裂的 key 能够减小后续的网络代价,某种程度上可以转换成减小路由表代价,但它们不完全等同.不同的是由于路由表仅仅记录了 key 和其所处节点的映射,因此它不关心 key 粒度的大小.但后续的网络代价是与网络传输中的元组数量相关,因此,不同粒度的 key 在拆分上对后续网络代价会有不同的影响.

4 均衡调整算法

本节首先描述均衡调整的总体架构,在总体架构中,将具体的迁移计划制定模块抽象出来;然后,通过对 key 的迁移方向与是否被拆分进行代价衡量,给出了实现单一代价优化和综合代价优化目标的解决方案.

4.1 迁移计划制定架构

并行节点均衡调整的本质即是将高负载节点中部分负载移动到低负载节点上.为了使均衡调整过程具有更好的灵活性,算法分两个步骤完成均衡迁移,见算法 2.首先,将负载节点中超出系统容忍度的负载迁移至一个中间集合(第 3 行~第 5 行);然后,将负载从中间集合装载到各低载的节点(第 6 行).均衡调整算法的输出结果是以聚合的元组为粒度的迁移计划 MP(第 7 行),表示为 MP={key,迁移数据量,所处负载节点,负载移动的目标节点},含义是将 key 的多少数据量从所处负载节点迁移至目标节点.

算法 2. 总体架构.

Input:节点集合$\mathcal{D}$中的负载节点,均衡度τ;

Output:迁移计划 MP. /*←:赋值;→:装载;(ϕ,ψ):$\phi\leqslant\psi$,ψ中的ϕ个;C:负载中间缓存集合.*/

1 $\mathcal{C}\leftarrow\varnothing$ //初始化中间结果集合

2 $UL\leftarrow(1+\tau)\cdot\overline{L}$ //计算系统不均衡的最大容忍度

3 **foreach** $d\in\mathcal{D}$ **do** /*在节点集合$\mathcal{D}$中的超载节点 d*/

4 **if** $L(d)>UL$ **then**

5 *UNLOAD*(d,UL,C)

6 *LOADING*(C,$\mathcal{D}$)

7 **return** MP

8 **function** *UNLOAD*(d,UL,C) /*将超载节点中的负载卸载至中间缓存集合 C 中*/

9 **while** $L(d)>UL$ **do**

10 $(\phi,\psi)\leftarrow$*UNLOADSELECT*(d) //从超载节点中选择卸载的负载

11 $(\phi,\psi)\rightarrow C$

12 *Del*(ϕ,ψ) in d

13 **function** *LOADING*(C,$\mathcal{D}$) /*将中间缓存集合 C 中的负载装载至低载节点中*/

14 **while** $C\neq\varnothing$ **do**

15 *LOADINGSELECT*(C)$\rightarrow\mathcal{D}$ //从中间缓存集合中选择装载到低载节点的负载

在算法 2 中,将超载节点的超出部分负载暂时卸载到中间集的函数 *UNLOADSELECT*(d)(第 10 行),以及将暂存于中间集 C 中的负载装载到各个低载节点的函数 *LOADINGSELECT*(C)(第 15 行),根据不同的迁移代价衡量(见第 3.3 节 O_1~O_4)会产生不同的迁移计划,即,迁移代价目标决定迁移内容.若要求系统快速制定使系统达到均衡状态的迁移计划,那么频繁地拆分 key 能够促使调整策略快速达到这一目的;而如果要求系统运行中的网络传输代价小,即,减少连接操作时连接数据的广播量,那么在卸载函数 *UNLOADSELECT*(d)和装载 *LOADINGSELECT*(C)中就不能随意地拆分 key,同时还要在函数 *LOADINGSELECT*(C)中,需尽量将 key 放入能够减少拆分 key 的节点中,即,key 增加合并动作;此外,追求廉价的迁移动作或较小的路由表代价也分别有其对移动 key 的不同选择.

在整体架构算法中,无论是单目标优化还是多目标的优化,卸载和装载函数只要做相应的 key 选择上的变化即可.表 2 分析了不同迁移动作与 key 拆分与否的状态对 O_1~O_4 代价衡量标准的利弊情况 O_1:迁移计划的制定速度,O_2:路由表代价,O_3:迁移代价,O_4:后续网络代价.⊕:促进,∅:无关,⊖:不利,⊗:可能相关).在表 2 中,无论 key 做何种迁移动作,使key趋于合并状态将有利于其迁移后的网络代价减少,这得益于聚合的key减少了待连接的数据广播量.

Table 2 Relation list

表 2 关系表

	回迁	移迁	迁出
合并	⊗⊕⊗⊕	⊗⊕⊗⊕	⊗∅⊗⊕
不变	⊗⊕⊗∅	⊗⊕⊗∅	⊗⊖⊗∅
拆分	⊕∅⊗⊖	⊕⊖⊗⊖	⊕⊖⊗⊖

由于篇幅限制,本文只分别介绍一种追求单一代价目标和一种追求整体代价目标的方法,分别命名为快速制定迁移计划算法 QMMP(quickly make migration plan)和整体迁移策略 MNRT(migration plan,split key, network coat and routing table).任意其他组合目标可以采用类似的策略实施.

4.2 快速制定迁移计划

按照元组级粒度随机地分发数据,可以使并行负载节点处于完美的均衡状态.QMMP(quickly make migration plan)算法选择了按照元组粒度级别来进行均衡调整,因此,其可使系统快速地达到均衡状态.从表 2 可知:不论是何种迁移动作,只要随意拆分 key 进行均衡调整,便有利于系统快速地制定出迁移计划.如算法 3 描述,无论是在将超载节点的负载卸载至中间缓存集合中,还是将负载从中间集合装载到低载节点中,QMMP 算法根据实际需求频繁地拆分 key 动作(第 6 行、第 7 行和第 15 行~第 17 行),进而使节点的负载达到预期均衡.

算法 3. QMMP.

Input:节点集合$\mathcal{D}$中的负载节点,均衡度τ;

Output:迁移计划 MP. /*←:赋值;→:装载;(ϕ,ψ):$\phi\leqslant\psi$,ψ中的ϕ个;C:负载中间缓存集合.*/

```
1  Line (1~7) in Algorithm 2
2  function UNLOADSELECT(d) do
3    χ←L(d)−UL
4    if L(k)≤χ then   //直接将整个 key 卸载至中间结果集 C
5      return (L(k),L(k))
6    else
7      return (χ,L(k))   //将 key 按需拆分,卸载至中间结果集 C
8  function LOADINGSELECT(C) do
9    foreach k∈C and d∈D do   /*将暂时存放在 C 的负载装载到低载节点至 L̄ */
10     if L(d)≤ L̄  then
11       χ← L̄ −L(d)
12       if L(k)≤χ then   //直接将整个 key 装载至低载节点
13         (L(k),L(k))→d   //从中间缓存集合 C 将 key 的全部负载量装载到低载节点 d
14         Del(L(k),L(k),C)   //从中间缓存集合 C 将 key 的全部负载删除
15       else   //将 key 按需拆分,装载至低载节点
16         (χ,L(k))→d   //从中间缓存集合 C 将 key 的部分负载量χ装载到低载节点 d
17         Del(χ,L(k),C)   //从中间缓存集合 C 将 key 的部分负载量χ删除
```

QMMP 在迁移负载过程中既无对 key 的排序,也没有对 key 的特定选择,其仅仅是按照各个节点的负载均衡需求量对 key 切分,因此,该算法的计算复杂度为常数级 $O(K)$(K 为 key 的个数).

4.3 综合目标制定迁移计划

除表 2 列举的不同 key 操作状态与迁移动作对 O_1~O_4 代价的影响之外,优先选择何种粒度的 key 进行迁移,对最终代价也有着不同的影响.例如:优先选择大粒度 key 迁移,那么移动较少个数的 key 既能使系统负载达到均衡,在一定程度上也可以控制路由表的增速,然而,这种选 key 方案可能会增加拆分 key 的频率,进而增大后续网络传输代价;反之,如果优先选择粒度较小的 key 进行迁移,路由表的增速则会变大.若优先考虑均衡调整过程中的迁移代价问题,则需优先选择迁移负载大而迁移携带数据量较小的 key 迁移(在 key 的当前负载与其携带的数据量关系不定的情况下).据此,本节接下来将给出考虑整体迁移代价的算法 MNRT.该方法将在不同阶段使用贪心的方式使系统达到均衡要求.

如算法 4 所示:MNRT 在第 4 行首先根据低载节点集合及超载节点中 key,判定迁移哪些 key 能够增加系统中 key 的合并操作,进而减少系统后续的网络代价及路由表代价;第 5 行~第 6 行中,MNRT 根据迁移代价的度量来选择需要移出的 key,进而减少系统在迁移过程中的迁移代价;随后,算法通过第 7 行~第 10 行将负载移出部分放至负载的中间缓存集合中,这里,MNRT 为了兼顾迁移计划的制定速度,将粒度较大的 key 进行拆分;在从中间缓存集合将负载装载至低载节点的过程中,MNRT 同样根据表 2 择优选取合适的 key 来移动,具体见算法的第 12 行及表 3.需要注意的是,当预装载 key 的负载过大时,系统并不立即将其拆分后装入低载节点,而是将该大粒

度 key 直接放入该低载节点,之后,将该节点当做超载节点来处理,如算法 4 第 13 行~第 16 行所示.这样做的原因有两个:1) 尽可能不拆分大粒度的 key,因其对后续网络代价影响较大;2) 低载节点有可能连续处于低载状态,此时,存在 key 被该低载节点锁定而无法与其他节点上的 key 聚合或进行回迁操作.

此外,在算法 4 拆分 key 的过程中,如果一条流的单个 key 大于一个节点的负载,则将其单独聚集于一个节点.同时,当两条数据流间存在数据的倾斜现象,算法 4 保持大流量端的数据不动,进而广播小数据量端的流数据,由于该方法类似于文献[3],本文不再赘述.

算法 4. MNRT.

Input:节点集合$\mathcal{D}$中的负载节点,均衡度τ;

Output:迁移计划 MP.

1 Line (1~7) in Algorithm 2

2 **function** *UNLOADSELECT*(*d*) **do**

3 $\chi \leftarrow L(d)-UL$

4 $k \leftarrow InThePriority(\gamma_1,\delta_1)\mapsto(\gamma_1,\delta_2)\mapsto(\gamma_1,\delta_3)\mapsto(\gamma_2,\delta_1)\mapsto(\gamma_2,\delta_2)$ according to $d,\mathcal{D}^*$. //根据低载节点集合$\mathcal{D}^*$以及表 2 中的代价关系,在超载节点 *d* 中选择符合上述优先级的 key.γ_1,δ_1代表第 1 行与第 1 列.

5 **if** $k=\varnothing$ **then**

6 $k \leftarrow GetTheKeyWithTheSmallest\ \frac{MC}{L(k)}$ in d

7 **if** $L(k)\leqslant\chi$ **then**

8 **return** $(L(k),L(k))$

9 **else**

10 **return** $(\chi,L(k))$

11 **function** *LOADINGSELECT*(*C*) **do**

12 $k,d \leftarrow InThePriority(\gamma_1,\delta_1)\mapsto(\gamma_1,\delta_2)\mapsto(\gamma_1,\delta_3)\mapsto(\gamma_2,\delta_1)\mapsto(\gamma_2,\delta_2)\mapsto(\gamma_2,\delta_3)\mapsto(\gamma_3,\delta_1)\mapsto(\gamma_3,\delta_2)\mapsto(\gamma_3,\delta_3)$ according to $C,\mathcal{D}^*$. //根据低负载节点集合$\mathcal{D}^*$及表 2 中的代价关系,在中间负载缓存集合 *C* 中选择符合这些优先级的 key

13 $(L(k),L(k))\rightarrow d$

14 $Del(L(k),L(k),C)$

15 **if** $L(k)\geqslant UL$ **then**

16 $UNLOAD(d,UL,C)$

4.4 算法分析

均衡保证方面,QMMP 方法以元组为粒度进行均衡调整,故其可以使并行负载节点达到完美的均衡状态.MNRT 在兼顾表 2 中列举代价衡量的同时,如算法 4 中的第 9 行、第 10 行,其进行选择性的拆分 key.因此,该方法也能够将系统调整至任意均衡状态.但选择性的拆分 key 增大了算法 MNRT 的计算复杂度,其计算复杂度可表示为 $O(K/N_D\times\ln(K/N_D)\times N_D\times K)$.其中:前半部分的代价 $O(K/N_D\times\ln(K/N_D))$是按照表 2 选取合适的 key 时 MNRT 付出的计算代价;后半部分 $O(N_D\times K)$是其在均衡保证部分付出的代价,即,最差的情况下 MNRT 会遍历所有 key 和所有节点来安置大粒度的 key,如算法 4 中第 16 行.因此,方法 QMMP 的计算复杂度可简化表示为 $O(K^2\times\ln(K/N_D))$.实际情况中,由于数据倾斜(长尾分布),MNRT 的计算代价会远远小于 $O(K^2\times\ln(K/N_D))$,这在本文实验部分中有具体体现.

很明显,上述各方法同样适用于诸如投影和选择等基础操作.对于聚合操作,本文的解决方案同文献[25],即拆分的 key 产生中间结果,最后汇总于其下游的输出节点.而本文的 MNRT 方法使用了对 key 按需拆分、尽量合并的策略,这在减轻汇总节点负载的同时,也将减小在消息保证机制下的系统延时.另外,上述算法支持对系统的横向扩展,即,能够支持系统对处理节点的增删.当系统新加入一个负载节点时,算法将该节点视为一个空白节点,进而根据新的基础路由(例如一致性哈希)将原负载节点上的负载转移至新加入的节点中.同理,本文方

法也同样支持系统删除负载节点的情形.

5 实 验

本节介绍实验的设置情况,包括实验中机器及并行数据流处理系统、使用的数据集、实验的操作语句、性能评估指标及各种方法的简介.最后给出各个方法的性能对比情况.

5.1 实验设置

5.1.1 实验环境

本文的所有实验部署在版本为 0.10.1 的 storm[21]系统上,该系统是由 23 台刀片机组成的集群,集群上的操作系统为 CentOS6.5.集群机器是主频为 2.00GHz 的 Intel Xeon 处理器(E5335/2.00GHz),内存为 16GB.方法实现代码均使用 Java JDK1.8.

5.1.2 数据集

本文使用 TPC-H 作为测试基准[28].数据倾斜方面,采用 Zipf 的数据分布.所有的生成数据在流入系统之前都需要预先处理,即:根据生成数据主外键关系,将其按照 Zipf 分布中的数据倾斜参数 z 调节数据分布倾斜度.默认情况下,本文采用的倾斜度为 z=0.8.此外,本文还使用 10GB 的微博数据统计话题热度情况.本文对所有方法使用了简单的散列哈希作为基础映射函数.另外,为了满足任意两个元组可进行运算的语义,在负载节点内的连接运算中,不使用索引方法组织 key.

5.1.3 操作任务

TPC-H 测试中选用等值连接(EQ_5)和非等值连接(B_{NCI})两种查询;在微博数据集上进行热门话题统计(TOPK).具体查询操作如下:

- EQ_5:SELECT*FROM Region *R*, Nation *N*, Supplier *S*, Lineitem *L*
  WHERE *R.regionkey=N.regionkey* AND *N.nationkey=S.nationkey* AND *S.suppkey=L.suppkey*;
- B_{NCI}:SELECT*FROM Lineitem *L*1, Lineitem *L*2
  WHERE |*L*1.*orderkey*–*L*2.*orderkey*|≤1 AND *L*1.*shipmode*='TRUK' AND *L*2.*shipinstruck*='NONE' AND *L*2.*Quantity*>48;
- TOPK:SELECT TOP 10, Topic COUNT(1)AS NUM FROM Social data
  GROUP BY Topic ORDER BY NUM DESC.

5.1.4 性能指标

实验中采用的性能指标定义如下:

- 节点使用个数:负载操作 DO 中根据内存负载定义使用的并行节点的个数,即,操作并行度 N_D 的值;
- 执行时间:操作处理完数据的时间;
- 节点平均接收数据量:为完成操作,每个节点接收到在 UO 和 DO 之间传输数据量的平均值,此指标直接反映了均衡调整后,系统的后续网络传输量;
- 制定计划的时间:系统计算出调整均衡的迁移计划时间.

5.1.5 实验对比方法

- QMMP:算法 3 提出的快速生成迁移计划的方法采用频繁的 key 拆分方法达到系统尽早均衡的目的.该方法频繁地拆分 key,从而使系统负载尽快达到均衡状态.然而,若 QMMP 对 key 的拆分过多,就会使得路由表变大及后续元组广播频率增加,从而增加了节点的负载均衡计算;
- MNRT:本文提出的使用贪心方法来尽量优化 O_1~O_4 衡量标准,即算法 4.该方法虽然不能像 QMMP 一样快速地制定迁移计划使系统达到均衡状态,然而由于综合考虑了后续元组的网络代价,其在 key 携带状态的均衡调整中能够表现网络代价小的优势;
- Readj[18]:该方法将 key 看做一个完整的粒度,在调整过程中不进行拆分.其均衡的调整方案是迁移时只考虑调整当前频率较大的 key,而小频率的 key 全部回迁.调整方法上使用节点两两配对,节点内 key 试

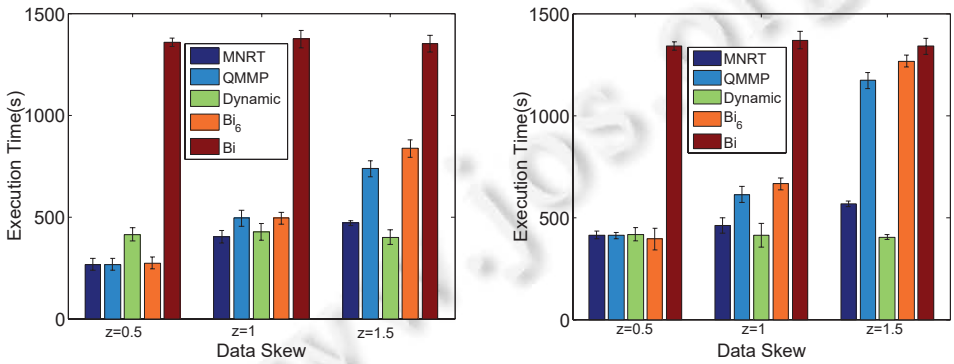
着交换来寻求均衡的暴力方法.这使得在海量 key 及均衡要求较高的情况下,其均衡调整的迁移计划制定时间大大增加;

- PKG[20]:将一个 key 拆分为更小的粒度后,有选择性地分发负载至不同并行节点上来避免负载倾斜问题.具体来说,PKG 将 key 拆分为两个部分,之后将拆分的两个部分置于不同的处理节点上,进而根据两个负载节点的负载,有选择性地分发后续到来的元组.本文使用了 PKG 提供的源码(https://github.com/gdfm/partial-key-grouping);
- Dynamic[22]:即矩阵模型的具体实现方法.Dynamic 将节点分布在矩阵的长和宽两个维度上进行数据流的连接操作,做连接的两条流通过这两个方向流入.然而,由于 Dynamic 设定矩阵内节点的个数为 2 的幂次方个,这使得矩阵形状的变化存在一定的局限性,从而浪费计算资源;
- Bi 和 Bi_6[19]是二分图模型处理数据流上的连接操作实现.Bi_6 表示在二分图的每组有 6 个子组,Bi 表示在二分图的各边没有子组划分.Bi 方法需要广播一侧的流数据到另一侧的所有处理节点上,这增大了网络代价;Bi_6 将处理节点分为子组,在一定程度上限定了该架构应对数据倾斜的特性.

5.2 实验结果

5.2.1 可扩展性-全历史连接操作

图 3 显示了将 12GB 的 TPC-H 数据处理完成的系统耗时情况.实验将元组装载的速度设置为最大,以使各个负载节点的计算能力充分发挥.图 3(a)展示了等值连接 EQ_5 在不同数据倾斜情况下的执行时间:当数据倾斜较小时,系统负载也较为均衡,因此 MNRT 和 QMMP 方法根据 key 定位使得数据的广播量减少,进而使得每个负载节点的负载较小;随着数据倾斜变严重,QMMP 对 key 的拆分动作增多,这使得后续连接操作中的元组广播频率增加,从而增加了节点的负载均衡计算,这一状况在图 3(b)中体现得尤为明显.对于二分图方法,当数据倾斜度严重时,Bi_6 方法中的分组受到了挑战,例如,当 z=1.5 时,大粒度的 key 占比增加,这造成了子组内广播的数据量增大,从而增加了负载节点的计算量,使系统的执行时间延长.Bi 方法由于不分子组,即全广播连接运算,这使得其架构内的负载节点的计算负载最大,从而导致其延时较高.基于矩阵模型的 Dynamic 方法随机分发元组,因此执行时间不受数据倾斜的影响;同时,由于该方法保证了两条数据流中的任意元组可碰面,等值或非等值的连接操作谓词不影响其性能,如图 3(a)和图 3(b)所示.



(a) EQ_5 查询语句 (b) B_{NCI} 查询语句

Fig.3 Execution time

图 3 执行时间测试

图 4 显示了各种方法在装载数据过程中的节点使用情况,Dynamic 方法限定节点个数是 2 的幂次方,每次扩展仅仅是简单的将过载节点由 1 个扩展为 4 个,势必造成资源浪费.本文的方法 MNRT 和 QMMP 能够根据装载量按需分配负载节点,因此,相对于 Dynamic 能够使用更少的节点资源.然而,从图上不难看出,二分图方法以优化内存为出发点也使用了较少的节点个数.具体来说:Bi_6 初始化时二分图各边(R/S)有 6 个子组,每个子组有一个

负载节点,随着装载量的变大,其子组内的节点个数按照装载的量扩容;Bi 初始化时二分图的每个边仅有一个节点,然而,由于其仅仅考虑以内存为优化目标,当 CUP 的计算缺乏时,其吞吐量会受到影响.这也体现在图 3 展示的执行时间上.

图 5 展示了各种方法在不同数据倾斜度情况下扩容的调整效率.如图 4 所示:Dynamic 使用了更多的并行节点个数,当系统扩容时,其需要迁移的数据量也是最多的,进而增大了调整的时间.QMMP 能够从元组粒度级别进行调整,该方法能够更快地计算出迁移计划,进而其调整的速度稍微领先于 MNRT.二分图方法中,Bi 在单条流对元组是随机存放的,所以它也更便利于系统中并行节点的均衡扩容.

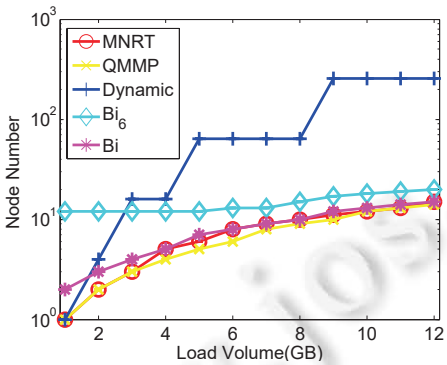


Fig.4 Parallel processing node number

图 4 并行处理节点使用个数

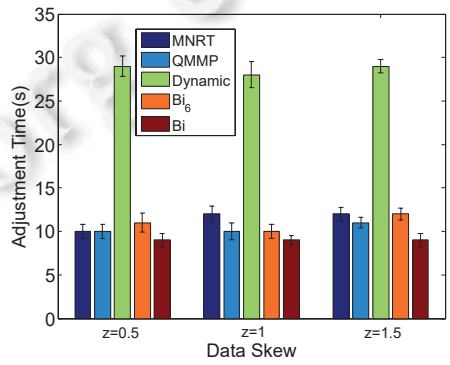


Fig.5 System balance adjustment time

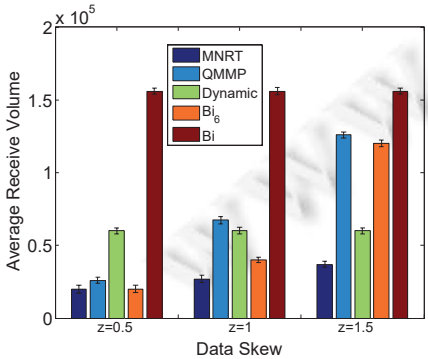
图 5 系统均衡的调整时间

由图 3~图 5 可以看出:本文提出的方法能够在使用相对较少节点的情况下,快速地处理完 12GB 的数据量.

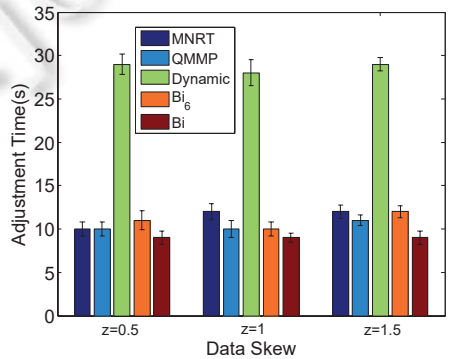
5.2.2 动态性-基于窗口模型的连接操作

为了进一步验证数据的倾斜及查询操作对性能的影响,本节使用窗口模型展示了不同的查询任务在不同倾斜数据分布情况下,每个负载节点平均接收到的元组个数.该组实验使用了 64GB 的 TCP-H 数据量,将窗口大小设置为 180s,元组发送速率约为每秒 3×10^5 条,该速度确保了每个负载节点的 CPU 可以满负荷.

图 6(a)和图 6(b)分别展示了 EQ_5 和 B_{NCI} 的查询任务在不同数据倾斜分布情况下及不同计算架构中每个节点平均接收到的元组数量.图 6 中,Dynamic 方法不受数据倾斜的影响,原因是该方法在矩阵的两个维度方向都是采用了随机分发方式.所以只要窗口内的数据量固定,无论数据是何种分布以及操作是何种连接,各个并行节点接收到的数据量是相同的.因此,该方法能够将计算负载均衡地分配到每各个负载节点上,但是该方法使用了较多的节点资源.



(a) EQ_5 查询语句



(b) B_{NCI} 查询语句

Fig.6 Subsequent network load

图 6 后续网络负载

本组实验结果从各个方法中节点的数据流入量测试各个节点的计算负载情况.MNRT 方法尽量将属于同一 key 的元组存储至同一个节点,其对不同粒度的 key 区别对待,这使得流数据中元组的路由方向性更好.即便在范围连接 B_{NCI} 操作中,MNRT 依然能够很好地控制元组数据的流向.然而,QMMP 方法频繁地拆分 key,虽然能够使系统快速制定出迁移计划,但是拆分 key 增加了连接流的广播量.在倾斜较小的情况下,对于等值连接 Eq5, Bi_6 方法根据路由能够更好地将连接请求分发至选中的子组,从而使节点接收到较少的处理连接请求.随着数据倾斜度加大,子组内的广播数据量加大.例如当 z=1 时,最大的 key 占据了一半的数据量,这一现象在范围连接操作 B_{NCI} 中尤为明显.Bi 由于是不分组的全广播,各个负载节点接收的连接请求在等值连接或范围连接中是同样的.

5.2.3 真实数据性能

本节使用 10GB 微博数据进行 TOPK 操作,以验证方法的有效性,该操作基于窗口模型进行.本组实验用于验证适用于聚合操作的均衡调整方法性能,由于 Dynamic 和 Bi 方法更倾向于做数据流的连接操作,故此组实验不涉及这两种方法.本组实验中,系统要求将不均衡容忍度调整至 0.05 以内.

图 7 展示了各种方法在不同均衡容忍度的要求下,制定迁移计划的效率.由于 PKG 方法中没有迁移动作,因此也没有迁移计划的制定动作.在图 7 中,QMMP 以元组为粒度拆分 key,无论要求的均衡容忍度为多大,它均能够较快地制定出迁移计划.MNRT 兼顾了整体系统代价,所以其迁移计划的制定比 QMMP 稍慢.Readj 方法由于要配对各个节点及节点内的 key 来尝试着交换不同节点中的 key 使系统均衡,这种方法计算复杂度较高,因此制定迁移计划时间更长.在图 8 中,各个方法都使用可靠性保证消息机制,同时,对于操作中的计数使用了并行的方式,这些并行计数的中间结果被定时(Storm 上的 tick 机制)汇总到一个聚合节点上.中间结果的定时汇总操作在拆分 key 更频繁的方法上体现出了更大的处理延时,从而降低了系统的整体吞吐量而使系统的处理速度变慢.Readj 由于对严格的均衡容忍度敏感,增长了迁移计划的制定时间,最终影响了整体系统的性能表现.

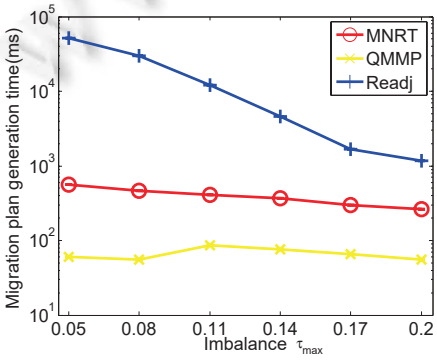


Fig.7 Migration plan generation time

图 7 迁移计划制定时间

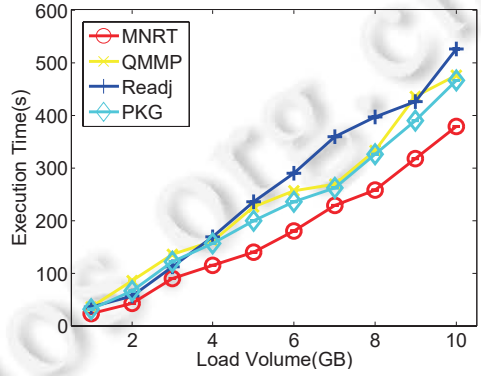


Fig.8 System performance

图 8 系统整体性能

6 总 结

在分布式流处理系统中,并行处理节点间不均衡负载会导致个别高负载节点处理效率降低而其他节点出现大量的空闲运算资源,从而降低了系统的整体利用率.本文通过对并行流处理系统在均衡调整过程中的代价分析,提出了一种对 key 按需拆分,尽量合并的均衡调整方法.该方法既方便于系统的均衡调整,又能减少系统在后续操作中对数据的广播动作,进而有效地减少了系统的额外负载,最终提升了系统的处理性能.然而,对于文中提到的多目标代价优化的实现,本文只给出了简单的利用分步贪心方法来获取各个代价标准的相对较优解,因此在未来的工作中,我们会进一步研究更优的方法来解决对目标代价的优化问题,将各个性能指标归一化,使系统达到更便捷、更优的均衡特性.

References:

[1] Gong XQ, Jin CQ, Wang XL, Zhang R, Zhou AY. Data-Intensive science and engineering: Requirements and challenges. Chinese Journal of Computers, 2012,35(8):1−16 (in Chinese with English abstract).

[2] Li JZ, Liu XM. An important aspect of big data: Data usability. Journal of Computer Research and Development, 2013,50(6): 1147−1162 (in Chinese with English abstract).

[3] Qian Z, He Y, Su C, Wu ZJ, Zhu HY. TimeStream: Reliable stream computation in the cloud. In: Proc. of the ACM European Conf. on Computer Systems. 2013. 1−14. [doi: 10.1145/2465351.2465353]

[4] Zhou Y, Ooi BC, Tan KL. Dynamic load management for distributed continuous query systems. In: Proc. of the IEEE Computer Society. 2014. 322−323. [doi: 10.1109/ICDE.2005.54]

[5] Zhu Y, Rundensteiner EA, Heineman GT. Dynamic plan migration for continuous queries over data streams. In: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. Paris, 2015. 431−442. [doi: 10.1145/1007568.1007617]

[6] Zhou YL, Ooi BC, Tan KL, Wu J. Efficient dynamic operator placement in a locally distributed continuous query system. LNCS 4275, 2006. 54−71. [doi: 10.1007/11914853_5]

[7] Fernandez RC, Migliavacca M, Kalyvianaki E, Pietzuch P. Integrating scale out and fault tolerance in stream processing using operator state management. In: Proc. of the 2013 ACM SIGMOD Int'l Conf. on Management of Data. 2013. 725−736. [doi: 10.1145/2463676.2465282]

[8] Jin CQ, Qian WN, Zhou AY. Analysis and management of streaming data: A survey. Ruan Jian Xue Bao/Journal of Software, 2004, 15(08):1172−1181 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/15/1172.htm

[9] Xu Y, Kostamaa P, Zhou X, Chen L. Handling data skew in parallel joins in shared-nothing systems. In: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. 2008. 1043−1052. [doi: 10.1145/1376616.1376720]

[10] Vitorovic A, Elseidy M, Koch C. Load balancing and skew resilience for parallel joins. In: Proc. of the IEEE Int'l Conf. on Data Engineering. IEEE, 2016. 313−324. [doi: 10.1109/ICDE.2016.7498250]

[11] Kwon YC, Balazinska M, Howe B, Rolia J. SkewTune: Mitigating skew in mapreduce applications. In: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. 2012. 25−36. [doi: 10.1145/2213836.2213840]

[12] Gufler B, Augsten N, Reiser A, kemper A. Load balancing in MapReduce based on scalable cardinality estimates. In: Proc. of the IEEE Int'l Conf. on Data Engineering. 2012. 522−533. [doi: 10.1109/ICDE.2012.58]

[13] Xing Y, Zdonik S, Hwang JH. Dynamic load distribution in the Borealis stream processor. In: Proc. of the Int'l Conf. on Data Engineering. IEEE, 2005. 791−802. [doi: 10.1109/ICDE.2005.53]

[14] Xing Y, Hwang JH, Cetintemel U, Zdonik S. Providing resiliency to load variations in distributed stream processing. In: Proc. of the Int'l Conf. on Very Large Data Bases. ACM Press, 2006. 775−786.

[15] Abadi DJ, Ahmad Y, Balazinska M, *et al*. The design of the Borealis stream processing engine. In: Proc. of the 2005 CIDR Conf., 2005. 277−289.

[16] Fang J, Zhang R, Fu TZJ, Zhang ZJ, Zhou AY, Zhu JH. Parallel stream processing against workload skewness and variance. arXiv preprint arXiv:1610. 05121, 2016.

[17] Shah MA, Hellerstein JM, Chandrasekaran S, Franklin MJ. Flux: An adaptive partitioning operator for continuous query systems. In: Proc. of the Int'l Conf. on Data Engineering. 2003. 25−36. [doi: 10.1109/ICDE.2003.1260779]

[18] Gedik B. Partitioning functions for stateful data parallelism in stream processing. Int'l Journal on Very Large Data Bases, 2014, 23(4):517−539. [doi: 10.1007/s00778-013-0335-9]

[19] Lin Q, Ooi BC, Wang Z, Yu C. Scalable distributed stream join processing. In: Proc. of the ACM SIGMOD Int'l Conf. 2015. 811−825. [doi: 10.1145/2723372.2746485]

[20] Nasir MAU, Morales GDF, Garcia-Soriano D, Kourtellis N, Serafini M. The power of both choices: Practical load balancing for distributed stream processing engines. In: Proc. of the IEEE Int'l Conf. on Data Engineering. IEEE, 2015. 137−148. [doi: 10.1109/ICDE.2015.7113279]

[21] Apache storm. http://storm.apache.org/

[22] Elseidy M, Elguindy A, Vitorovic A, Koch C. Scalable and adaptive online joins. Proc. of the VLDB Endowment, 2014,7(6): 441−452. [doi: 10.14778/2732279.2732281]

[23] Nasir MAU, Morales GDF, Kourtellis N, Serafini M. When two choices are not enough: Balancing at scale in distributed stream processing. In: Proc. of 2016 IEEE 32nd Int'l Conf. on Data Engineering, 2016. 589−600.

[24] Fang JH, Zhang R, Wang XT, Fu TZJ, Zhang ZJ, Zhou AY. Cost-Effective stream join algorithm on cloud system. In: Proc. of the 25th ACM Int'l on Conf. on Information and Knowledge Management. ACM Press, 2016. 1773−1782. [doi: 10.1145/2983323.2983773]

[25] Okcan A, Riedewald M. Processing theta-joins using MapReduce. In: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. 2011. 949−960. [doi: 10.1145/1989323.1989423]

[26] Fang JH, Wang XT, Zhang R, Zhou AY. Flexible and adaptive stream join algorithm. In: Proc. of the Asia-Pacific Web Conf. Springer Int'l Publishing, 2016. 3−16. [doi: 10.1007/978-3-319-45817-5_1]

[27] Bruno N, Kwon YC, Wu MC. Advanced join strategies for large-scale distributed computation. Proc. of the VLDBEndowment, 2014,7(13):1484−1495. [doi: 10.14778/2733004.2733020]

[28] The TPC-H benchmark. http://www.tpc.org/tpch

附中文参考文献:

[1] 宫学庆,金澈清,王晓玲,张蓉,周傲英.数据密集型科学与工程:需求和挑战.计算机学报,2012,35(8):1−16.

[2] 李建中,刘显敏.大数据的一个重要方面:数据可用性.计算机研究与发展,2013,50(6):1147−1162.

[8] 金澈清,钱卫宁,周傲英.流数据分析与管理综述.软件学报,2004,15(8):1172−1181. http://www.jos.org.cn/1000-9825/15/1172.htm

房俊华(1985－),男,河南沈丘人,博士生,主要研究领域为分布式流处理.

王晓桐(1994－),女,硕士,主要研究领域为分布式流处理.

张蓉(1978－),女,博士,副教授,主要研究领域为分布式计算.

周傲英(1965－),男,博士,教授,博士生导师,CCF杰出会员,主要研究领域为Web数据管理,数据密集型计算,内存集群计算,分布事务处理,大数据基准测试和性能优化.