一种基于动态拓扑的流计算性能优化方法及其在Storm中的实现

陆佳炜[1],吴 涵[1],陈 烘[2],张元鸣[1],梁倩卉[3],肖 刚[1]
(1. 浙江工业大学计算机科学与技术学院,浙江杭州 310023; 2. 阿里巴巴基础架构事业部大数据计算与服务团队,浙江杭州 310011; 3. 南洋理工大学计算机科学与工程学院,新加坡 637457)

摘 要: 响应性和稳定性一直是流式计算中两个至关重要的问题,而流计算系统在过载时常常表现出数据计算延迟增加和拓扑不稳定的现象,无法适应数据负载的动态变化.针对这一问题本文研究提出了一种基于动态拓扑的流计算性能优化方法,主要包括:(1)动态逐级反压:拓扑中的任务可以根据当前自身负载情况,动态调整上游向其发送数据的速率.(2)无状态拓扑数据重放:拓扑不维持数据的计算状态,尽可能地实现数据容错.(3)自适应拓扑替换:在拓扑不暂停的情况下对任务并发度进行自发调整.(4)延迟持久化队列:拓扑中对磁盘的IO读写被延迟到数据处理之外,减缓IO高频阻塞对流计算系统的影响.本文在Apache Storm中实现了以上四种方案,性能测试结果表明优化后的流计算系统与Storm默认实现相比,不仅增强了大数据动态匹配能力,而且在最优情况下改善了17%的吞吐量,并提升了约20%的数据处理速度.

关键词: 数据流拓扑; 流计算; 大数据; 流计算系统; 性能优化
中图分类号: TP311.13 **文献标识码:** A **文章编号:** 0372-2112 (2020)05-0878-13
电子学报URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2020.05.007

A Performance Optimization Method Based on Dynamic Topology for Stream Computing and Its Implementation in Storm

LU Jia-wei[1], WU Han[1], CHEN Hong[2], ZHANG Yuan-ming[1], LIANG Qian-hui[3], XIAO Gang[1]
(1. *Department of Computer Science and Technology, Zhejiang University of Technology, Hangzhou, Zhejiang* 310023, *China*;
2. *Team of Big Data Computing and Service, Department of Infrastructure Business, Alibaba, Hangzhou, Zhejiang* 310011, *China*;
3. *School of Computer Science and Engineering, Nanyang Technological University, Singapore* 637457, *Singapore*)

Abstract: Responsiveness and stability have always been two important problems in stream computing. However, as the scale of data being processed in real-time has increased, along with an increase in the data processing latency and topology instability of stream computing, many limitations of stream processing system have become apparent. Aiming at these problems, we present a performance optimization method based on dynamic topology for stream computing: (1) Dynamic step-by-step backpressure: the task in the topology can dynamically adjust the rate of upstream data transmission according to the current load. (2) Stateless topology data replay: topology can achieve data fault tolerance autonomously without maintaining the calculation of data state. (3) Adaptive topology replacement: no need for topology to suspend, the system can adjust the task concurrency spontaneously. (4) Delayed persistent queue: it delays the IO reading and writing in the disk out of the data processing, which mitigates the impact of IO high-frequency blocking in stream computing system. In this paper, the four methods are implemented in Apache Storm. The experimental results show that the optimized system not only enhances the dynamic matching capability of big data, but also achieves 17% higher throughput and 20% better data processing speed in the best case.

Key words: data stream topology; stream computing; big data; stream computing system; performance optimization

收稿日期:2019-07-02;修回日期:2019-11-13;责任编辑:马兰英
基金项目:国家自然科学基金(No.61976193);浙江省自然科学基金(No.LY19F020034);浙江省重点研发计划项目(No.2018C01064)

1 引言

随着移动互联网和物联网的到来,大数据进入了高速发展阶段. 各个公司为了应对大数据的挑战,急切地需要大数据技术解决生产实践中的问题. 例如,在每一分钟内,Facebook 用户分享近 250 万条内容;Twitter 用户推文近 300 万次;Instagram 用户发布近 220 万张新照片;YouTube 用户上传新的视频内容长达 72 小时;苹果用户下载应用程序近 50000 次[1]. 大数据处理的传统平台 Hadoop 和它的编程模型 MapReduce[2] 被设计用于静态数据的批量处理,而大量数据在不同时空中流转,呈现出鲜明的流式特征,需要更高的数据实时计算能力和使用能力[3],这使得流式大数据处理与传统批量大数据处理在数据计算的要求、方式等方面有着明显的不同,因而越来越多公司开始专注于流式计算应用.

从社交网络资讯到广告数据处理引擎,流计算系统在当今工业中被广泛使用,如 Apache Storm①,Twitter Heron②,Apache Flink③,Spark Streaming④,Samza⑤ 等. 在这些系统中,数据的产生完全由数据源确定,由于不同的数据源在不同时空范围内的状态不统一且易动态变化,导致数据流的速率呈现出了突发性的特征[3],容易导致过载的发生,此外网络拥塞、资源利用率高、干扰、异质性、IO 高频阻塞等也是发生系统过载的重要原因之一. 因此,在大数据流式计算系统中,过载是常见且难以避免的.

对于实时性系统,系统的响应性和稳定性是至关重要的. 为此,本文设计了一种基于动态拓扑的流计算性能优化方法,主要包括:(1)动态逐级反压. (2)无状态拓扑数据重放. (3)自适应拓扑替换. (4)延迟持久化队列. 我们在 Storm 中实现了该优化方法,实验结果表明优化方法可以有效改善 Storm 的响应性和稳定性,并弥补 Storm 拓扑在灵活性方面的不足,在数据流量高峰时期能够实时适应数据增长的需求,同时实现对系统资源的动态调整.

2 相关工作

在过去的几十年中,国内外关于流式计算的研究取得了许多令人瞩目的进展[3~9]. 例如文献[7]将流式计算与机器学习模型相结合,设计了一种面向医疗大数据的健康状态实时预测系统. 文献[8]将流式计算引入分布式数据管理系统中,以此提升分布式数据管理系统的事务处理效率和数据分析能力. 文献[9]利用流式计算的数据处理优势,对气象数据进行高效的分析与预测. 其中,流计算系统作为流式计算的运行载体呈现出多样化的发展趋势,总的来看流计算系统的发展可以划分为三代[10]. 第一代流计算系统通常为单机版,所以系统的处理功能会受到一定的限制;第二代流计算系统在第一代系统的基础上发展出分布式体系结构,并且在容错性和稳健性等方面有了很大的提高;第三代流计算系统的诞生是由云计算技术促成的,与前两代系统相比,第三代流计算系统不仅功能强大,而且具有更佳的可扩展性. 目前比较流行的流计算系统有 Apache 的 Storm[11], Yahoo 的 S4[12], Facebook 的 Data Freeway and Puma[13], Microsoft 的 TimeStream[14] 等.

长期以来,流计算系统的响应性和稳定性一直是学术界研究的重点对象. 文献[15]从数据计算延迟增加的原因展开研究,并且对尾延迟处理技术进行了详细的介绍和分析. 文献[16]对比分析了硬件、操作系统、应用程序等因素对尾延迟所产生的影响. 同时,许多研究学者也从数据流的连接和尾延迟等角度提出了相关的优化方案,文献[17]提出了一种基于特定操作的并行流连接算法,能够有效应对数据流的突发性变化. 文献[18]提出了一种低延迟握手算法,使计算设备可以在不影响数据吞吐量的情况下减缓延迟. 文献[19]针对尾延迟提出了一种基于延迟的负载均衡算法,用于提升数据流处理效率,并在 Storm 中进行了相应的算法实现. 此外,在最新的研究工作中,文献[20]通过分析研究流计算系统中响应时间和数据流到达率之间的数学关系,提出一种运行时调度模型来降低系统响应延迟,提高整体吞吐量. 文献[21]从系统架构、应用场景等方面对比分析了目前最为主流的五种流计算系统,认为解决数据处理时延和系统稳定性仍是流计算系统优化的关键方向. 文献[22]不仅对比分析了流计算系统之间的优缺点,并且在此研究基础上设计出一种部署更加便捷的流计算框架,它可以在运行时自适应调整计算资源,使得整体计算能力更为稳健高效.

Apache Storm 是一款典型的第三代流计算系统,自诞生之日起就不断受到业界与学术界的关注. 回顾 Storm 的性能优化研究,由于 Storm 默认调度器没有很好的考虑到组件实例(Task)间通信开销的问题,因此,对 Storm 默认调度器的改良一直是 Storm 性能优化研究的热点,目前已经有不少学者对此提出了优化方案. 其中,文献[23]为了更好的分析 CPU、内存、网络等带宽等因素对 Storm 任务迁移的制约影响,设计了一种运行时任务迁移模型,以此寻求 Storm 任务的最优迁移策略,保证 Storm 各工作节点的负载均衡. 文献[24,25]也分别提出了流量感知的调度策略和资源感

① Apache Storm:http://storm.apache.org/
② Twitter Heron:https://github.com/apache/incubator-heron
③ Apache Flink:http://flink.apache.org/
④ Spark Streaming:https://spark.apache.org/
⑤ Apache Samza:http://samza.apache.org/

知的调度策略,并在 Storm 默认调度器上实现了改进. 虽然上述的这些解决方案都在一定程度上减少了节点之间的通信开销,但是暂停拓扑以进行资源的重新分配仍然还是目前最主要的解决方案,由于重新分配过程中系统处于不可用状态,这又可能会导致更长的延迟和数据的丢失. 为了在集群运行时提前应对数据过载的问题,文献[26]提出在 Storm 平台上构建一种新颖的预测调度框架. 通过拓扑结构和运行时的统计信息进行预测分析并给出最佳的调度方案. 但是,这种技术本质上更依赖于获取所有服务器节点上的最新负载信息,在数据快速移动且以毫秒为单位执行的流处理系统中,持续地监控服务器节点负载会严重加剧系统负荷.

此外,在实时数据流中随着流速的突发性变化,会导致 Storm 中的 Tuple 处理失败,这一问题也一直是 Storm 性能优化研究的关注点. Heron[27] 是 Twitter 推出的流式计算引擎,它改善了 Storm 的许多瓶颈,如用反压策略解决拓扑过载时动态调整速率的问题. 当数据迅速流入拓扑,它通过降低数据源的发射速率使得拓扑稳定;然而,Heron 的反压策略只是简单地对源头进行限流,这种处理方式不仅不够灵活,而且很可能导致整体拓扑数据计算延迟. 与此类似,Storm 在 1.0 之后的版本①中,也引入了一种自动反压(Automatic Back Pressure)机制,当某个执行器发现数据过载时,整个拓扑都会进入反压状态,Spout 也会相应的降低发送速率,但是这种方法仍会对其他正常运行的组件实例造成不必要的开销.

与上述研究成果相比,本文提出的性能优化方法的优点主要体现在:

(1)处于过载状态的拓扑组件直接向上游组件进行逐级反压,有效的抑制了负载振荡,保证拓扑更加高效灵活的运行.

(2)能够自适应进行拓扑的缩/扩容替换,并且替换过程对用户透明,不需要终止程序的运行.

(3)尽最大限度保证对数据的重放处理,且处理方法比 Storm 的默认数据重放机制更加简便高效.

(4)通过构造延迟持久化队列来减缓 IO 高频阻塞对流计算系统的影响.

3 相关概念

在流式计算中,数据以流的形式持续地到来,数据的生成可以看成是一连串连续发生的离散事件,这些离散的事件以时间轴为维度形成了数据流. 不同于传统的批量处理,数据流指的是由一个或多个数据源持续生成的数据,伴随着这些数据流的是数据的操作和分析. 以 Storm 为例,Storm 的计算任务可以通过拓扑结构来描述,拓扑是一个有向无环图(如图 1 所示). 数据流在图中流过后,会产生相应的计算结果.

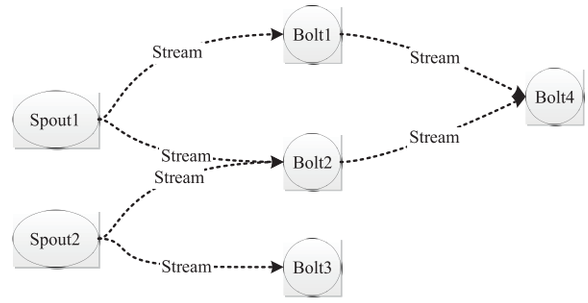


图1 有向无环图

Storm 是一个开源的分布式流计算系统,整个系统采用典型的主从(Master-Slave)架构方式,由运行 Nimbus 服务的主节点和运行 Supervisor 服务的工作节点组成. 其中 Nimbus 与 Supervisor 之间相关的调度协调信息都存储在 Zookeeper② 集群中,所以 Nimbus 和 Supervisor 进程都是快速失败(fail-fast)和无状态(stateless)的,表 1 介绍了 Storm 定义的相关抽象概念.

表 1 Storm 的抽象概念

抽象概念	概念描述
拓扑(Topology)	拓扑是表示分布式计算结构的有向无环图(DAG),由 Spout 和 Bolt 组成,通过流分组实现 Spout 和 Bolt 之间的相互关联. 拓扑会一直在集群中运行,除非显式地终止
源组件(Spout)	是拓扑中的数据发射源,用于接收外界数据并为拓扑提供数据
处理组件(Bolt)	Bolt 是拓扑中的数据处理单元,Bolt 接收到数据后进行相应的计算如过滤、聚合、存储等,也可以继续向外发送数据
组件(Component)	Spout 和 Bolt 统称为拓扑的组件
元组(Tuple)	Tuple 用来包装实际处理的数据,是一次消息传递的基本单元,在拓扑的 Component 之间流动,Tuple 从 Spout 发起,在汇聚节点结束,汇聚节点一般是位于拓扑末端的 Bolt
工作进程(Worker)	Worker 是一个物理进程,一个 Topology 会分配到一个或多个 Worker 上,Worker 的生命周期由工作节点上的 Supervisor 进程管理

① https://issues.apache.org/jira/browse/STORM-886

② Apache ZooKeeper:https://zookeeper.apache.org/

续表

抽象概念	概念描述
执行器(Executor)	Executor 是流计算执行任务的执行线程,Executor 在 Worker 中运行,是 Task 所在的直接容器,一个 Executor 运行一个或多个 Task
任务(Task)	Task 是 Component 的运行实例,一般情况下,一个 Component 在运行时会有多个 Task 实例,其数量由并发度来决定
操作(Operate)	Operate 是指 Tuple 在每一个 Component 中进行响应的行为,如过滤,聚合,替换等
流分组(Stream Grouping)	Stream Grouping 是指 Task 接收或发送 Tuple 的策略
消息树(Message Tree)	Message Tree 是指 Tuple 在拓扑中的流动路径

4　设计

拓扑计算是流式计算的核心,本节给出了流计算应用系统在拓扑执行过程中常见的问题,并对 Storm 的拓扑处理机制及缺陷进行了阐述,进而详细介绍了本文的动态拓扑优化方法,其核心实现机制包括:(1)动态逐级反压.(2)无状态拓扑数据重放.(3)自适应拓扑替换.(4)延迟持久化队列.

4.1　动态逐级反压

在 Storm1.0 之前的版本中,拓扑的 Tuple 由上游发送组件主动推送给下游接收组件,上游节点往往不会过多地考虑下游节点的负载情况、工作状态等因素,如果遇到下游节点因过载无法处理数据的情况,Storm 采用了 fail-fast 策略,即通过确认器(Acker)的开启状态来决定 Tuple 的丢弃或重新发送. 而在 Storm1.0 之后的版本中,引入了自动反压机制来监控下游接收组件的水位变化情况,当某一下游接收组件的水位达到阈值后,整个拓扑中便会进入反压状态,Spout 也会随之降低发送速率.

本文采用动态逐级反压来调整 Tuple 流过组件的速率. 此策略根据 Task 的当前自身负载情况来调整上游向其发送数据的速率,主要有以下特点:(1)无需设置额外的监控节点.(2)当前组件只负责向上游反压.(3)平滑反压抑制负载振荡.

拓扑中的组件如图 2 所示,对于每个组件 C(Spout 或者 Bolt),在数据流 t 输入后,触发事件 f_v,f_v 对 t 进行相关操作后产生新的数据流 t'并发送给下一个组件. 接下来 f_v 会监听下游组件是否发送反压信号 p,根据 p 将状态从 S 改变到 S'(若未收到 p 则状态不变,$S=S'$).

当 Task 的输入队列负载达到最大阈值时,对各上

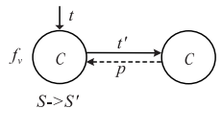


图2　任务拓扑组件

游节点执行动态逐级反压算法(算法 1)进行反压,收到反压信号的上游 Task 会减缓向下游发送数据的速率,以减轻下游的 Task 负载,但这可能会导致上游的队列阻塞,因此上游可能会继续向更上游反压,在最坏情况下,反压直到拓扑的 Spout 才结束. Task 每一次反压都会将速率下降至当前速度的 $1/v'$,采用定时器(timeSchedule)并设置其灵敏度为 sensitivity,以保证了在规定时间内针对某一个 Task 的反压信号至多发送一次,其中灵敏度 sensitivity 的计算公式如(1)所示:

$$\frac{(QML - queueLength)}{2 \times T.V} \tag{1}$$

其中,QML 表示 Task 输入队列总体长度,$queueLength$ 表示当前 Task 缓存队列长度,$T.V$ 表示当前 Task 在反压发生之前的数据发送速率. 若反压信号发送失败,算法会将 Task 重新加入反压队列.

算法 1　动态逐级反压算法

输入:拓扑任务集合 T,$Task_i$ 缓存队列总体长度 QML,定时器灵敏度 $sensitivity$,任务队列负载因子 $loadFactor$,$Task_i$ 缓存队列当前长度 $queueLength$,$Task_i$ 上游节点的状态 S_i

输出:无

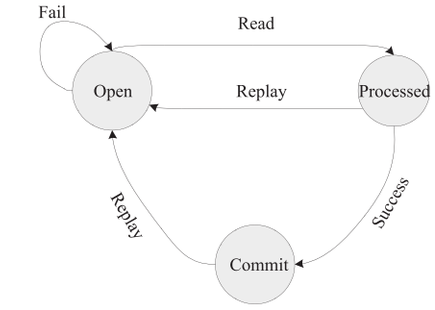
```
1. BEGIN
2.    Get sensitivity for system;
3.    Get loadFactor for system;
4.    FOR task_i in T DO
5.       Get queueLength, S_i of task_i;
6.       Get timeSchedule of task_i;
      //If queueLength reaches a threshold value and timeSchedule is null
7.       IF queueLength > loadFactor * QML AND timeSchedule = = null THEN
          // task_i sends a back pressure signal p to S_i and sets sensitivity
8.          S_i. State = timeSchedule. schedule({send a back pressure signal
            p to S_i}, sensitivity);
9.          IF S_i. State = = true THEN//If the back pressure signal p is sent
            successfully
10.            IF S_i. V_s = = null THEN
11.               S_i. V_s = S_i. V;//The back pressure needs to record the ini-
                  tial speed S_i. V_s at first time
12.            END IF
13.               S_i. V = S_i. V/v';//Set S_i. V becomes 1/v' of the
                  current speed
14.            cancel the timeSchedule of task_i;
15.         ELSE//If the back pressure fails, then add task_i back to T
16.            add task_i to T;
```

```
17.        END IF
           //If queueLength reduced to a minimum threshold for a sensitivity second
18.      ELSE IF queueLength < (1-loadFactory) * QML AND last sensitivity THEN
19.        task_i sends a back pressure cancel signal p' to S_i;
20.        IF S_i. State = = true THEN//If S_i is in the back pressure state
21.          S_i. V = S_i. V * v';//Set S_i. V becomes v' times of the current speed
               //If S_i. V returns to the initial speed S_i. V_s, then S_i cancels the back pressure state
22.          IF S_i. V_s = = S_i. V THEN
23.            S_i. State = = false;
24.          END IF
25.        END IF
26.      END IF
27.    END FOR
28. END
```

算法的运行效果是,当某一 Task 过载,上游的 Bolt 会收到反压信号并减缓向下游发射的速率,更多的资源会被用来处理当前的数据,避免因阻塞、数据超时、重发等导致的数据计算延迟增加. 当 Task 的负载降低至最小阈值且持续 *sensitivity* 秒,Task 会向上游发送取消反压信号,上游 Task 收到信号后首先会检查自身是否处于反压状态,若是则会恢复上一次的发射速度,只有当组件恢复至初始速度时才会消除反压状态. 迭代地进行反压/取消反压和使用定时器的原因都是为了抑制反压导致的负载振荡.

4.2 无状态拓扑数据重放

Tuple 丢失或 Tuple 处理时间超过拓扑的规定时间都会导致 Tuple 重放[3],但 Tuple 的频繁重放会导致 Tuple 对系统资源的长时间占据进而造成拓扑过载. Storm 的处理方法是在 Spout 上层维护一个待发送队列,队列中的一条 Tuple 被发送出去之后不会被立即删除,而是维持一种 Processed 状态,直到拓扑处理完成的信号到达. 如图 3(*a*)所示,若 Tuple 在计算过程中失败,拓扑可以重发处于 Processed 状态的 Tuple,以实现数据的容错. 但在大规模实时流计算中保存大量数据的计算状态无疑会增加系统的负载和维护的复杂性,为简化数据重放,减少拓扑计算延迟,我们提出无状态拓扑数据重放方法,对 Tuple 重放进行了优化,其数据状态机如图 3(*b*)所示.



(*a*)默认方法数据状态机

(*b*)无状态拓扑数据重放数据状态机

图3 默认方法数据状态机与无状态拓扑数据重放数据状态机

我们在 Spout 上层维护三个并行的待发送队列,分别为 Eden 队列,From 队列和 To 队列,其中 Eden 为原始队列区域,From 和 To 为复制队列区域,无状态拓扑数据重放方法流程如图 4 所示.

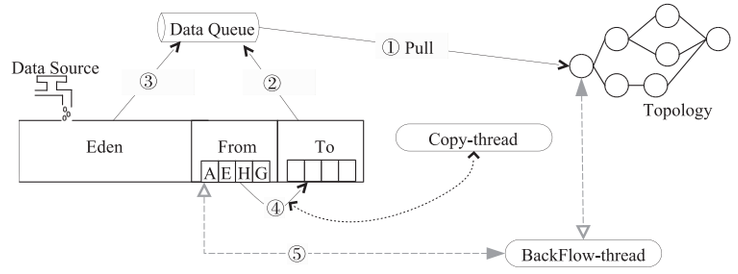


图4 无状态拓扑数据重放方法

①拓扑的 Spout 从数据队列 Data Queue 中拉取数据,Data Queue 连接待发送队列并根据以下策略(步骤②,③,④)读取数据;

②若 To 队列非空,读取 To 队列的数据,直到 To 队列为空;

③若 To 队列为空,读取 Eden 队列,进行步骤④;

④将 To 队列与 From 队列替换,效果同等于将 From 中的数据复制到 To 再清空 From,该步骤由 Copy-thread 线程负责执行. 重复步骤②,③,④;

⑤被读取 Tuple 的状态由“Open”状态变为“Commit”状态,拓扑不跟踪数据的计算状态. 若 Tuple 在拓扑中处理失败,Backflow-thread 线程(与步骤②并行)会

将处理失败的 Tuple 写回到 From 队列,等待 Data Queue 重放该 Tuple.

以上过程持续到所有数据处理完成. 步骤①,②,③属于数据读取,步骤④为复制算法,步骤⑤为数据回流,它们分别在不同的线程中并行执行,由于 To 队列的优先读取与复制算法的存在,尽可能地保证了数据的容错. 此外,虽然在复制期间(步骤④)To 队列不可读,From 队列不可写,但实际上该复制方法只是将指向 From 与 To 的指针进行替换,并清除 From 中的数据,此过程只需执行少量磁盘读写操作,因此算法执行速度得以保证.

4.3　自适应拓扑替换

拓扑过载时,除了针对 Tuple 重放进行优化,本文还提出了一种对用户透明,系统无暂停且不受 Task 并发度限制的拓扑替换方案——自适应拓扑替换. 其核心思想是收集多段时间窗口内拓扑中因超时而失败的 Tuple 数量,计算失败 Tuple 数量的均值 $E(M)$ 与加权离散率 S^2,根据均值和离散率选择是否用新的拓扑进行替换. 计算公式如式(2)、式(3)所示,其中 W_i 表示权值,权值与时间窗口相关,距离当前时间窗口越久的权值越低,X_i 表示 i 时间窗口内计算失败 Tuple 数量:

$$E(M) = \frac{\sum_{i=1}^{n} X_i}{n} \tag{2}$$

$$S^2 = \frac{\sum_{i=1}^{n} W_i \times (X_i - E(M))^2}{n} \tag{3}$$

S^2 值越大,表示当前拓扑越不稳定. 若在当前时间窗口内失败 Tuple 数量大于 $E(M)$,且 S^2 大于阈值 C,则进行扩容;若在当前时间窗口内失败 Tuple 数量小于 $E(M)$,且 S^2 大于阈值 C,表示当前拓扑有可能资源过剩,则进行缩容.

本文使用 2 次幂的缩扩容方式,*oldcap* 与 *newcap* 分别表示拓扑变化前后的并发度,*newcap* 的值由计算式(4)给出:

$$newcap = \begin{cases} oldcap \ll 1, & \text{if scale up} \\ oldcap \gg 1, & \text{if scale down} \end{cases} \tag{4}$$

$hash(i)$ 与 $newindex(i)$ 分别表示拓扑变化前后的索引值,i 表示哈希表中的记录序号,$newindex(i)$ 的值由计算式(5)给出(式中⊕表示异或运算符):

$$newindex(i) = \begin{cases} hash(i), \\ \quad \text{if scale up and } i = 0, \cdots, (newcap/2) - 1 \\ hash(i - newcap/2) \oplus newcap, \\ \quad \text{if scale up and } i = newcap/2, \cdots, newcap - 1 \\ hash(i), \\ \quad \text{if scale down and } i = 0, \cdots, newcap - 1 \end{cases} \tag{5}$$

以扩容为例,如图 5 所示,(old)表示扩容前的索引位置的示例,(new)表示扩容后的索引位置的示例,其中 Len 为哈希表长度,扩容后的哈希表长度分别与 keyA、keyB 进行异或运算,进而得出新产生的两种索引位置 keyC 与 keyD.

根据上述原理,可知在平均情况下该方法能减少一半的索引位置变换,从而减少 Task 调整的时间. 基于上述公式,算法 2 给出了自适应拓扑替换算法的实现,其中 *RingBuffer* 是一个循环使用的队列,存储最近一段时间失败 Tuple 的数量,它记录的时间区间是[当前时间-有效时间窗口数 * 时间窗口长度,当前时间].

算法 2　自适应拓扑替换算法

输入:当前时间窗口 *CTW* 的 Tuples 失败数量 *failNum*,存储失败 Tuple 数量的环队列 *RingBuffer*,*RingBuffer* 中有效的时间窗口个数 *A*,当前拓扑 *CT* 的并发度为 *currentParallelism*,时间窗口权值 $Wt(i)$

输出:无

```
1.  BEGIN
2.    FOR CTW DO
3.      Get failNum of CTW;
4.      Get A of RingBuffer;
5.      Get currentParallelism of CT;
6.      FOR i in {1,2,3,4,……,A} DO
7.        E(M) = E(M) + RingBuffer[i];
8.      END FOR
9.      E(M) = E(M)/A;//Get the mean of the number of failed tuples E(M)
10.     FOR i in {1,2,3,4,……,A} DO
11.       S² = S² + (RingBuffer[i]-E(M))² * Wt(i);
12.     END FOR
13.     S² = S²/A;//Get weighted dispersion rate S²
          //If failNum exceeds E(M) and S² exceeds threshold C
14.     IF failNum > E(M) AND S² > C THEN
          //The concurrency of the new topology is twice that of the original topology
```

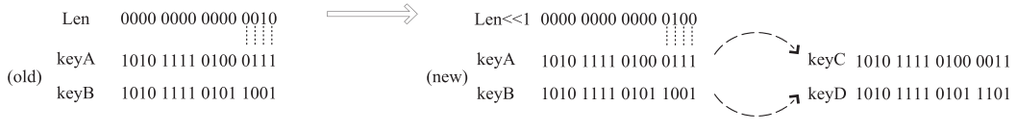


图5　扩容Rehash示例

```
15.        replacetopology (currentParallelism <<1);
             //If failnum does not exceed E(M) and S^2 exceeds threshold C
16.      ELSE IF failNum < E(M) AND S^2 > C THEN
             //The concurrency of the new topology is one half of the original topology
17.      replacetopology(currentParallelism >>1);
18.      END IF
19.    END FOR
20. END
```

该算法在第 15 步进行的是扩容替换,在 17 步进行的是缩容替换,等原拓扑处理完 Message Tree 中的 Tuple 后,新拓扑会执行替换. 替换过程对用户透明,由程序自动监测并进行替换,且替换过程中程序不会暂停(新拓扑已分配好 Worker,等待旧拓扑停止后即启动). 结合 4.2 节的无状态拓扑数据重放方法,原拓扑中失败的 Tuple 会在 To 队列由新拓扑来处理,因而不存在因频繁重发导致原拓扑无法被替换的情况.

4.4 延迟持久化队列

大部分数据在拓扑计算完成后被直接丢弃,剩余数据被持久化保存到磁盘中[3],但是流式大数据的不确定性决定了需要保存到磁盘的数据量在不同时空有较大的波动,这就需要数据持久层提供较好的写入性能. 快速写入通常是分布式 NoSQL 数据库(如 HBase①,Cassandra②)的强大特性,然而,对于处理快速数据流的分布式应用程序,负责持久化的数据库往往会成为性能瓶颈,从而导致流计算系统整体拓扑延迟增加.

延迟持久化队列(算法 3)是上述问题的优化方案,其将持久化操作分割为两部分流程:(1)在持久层上层维护一个内存读写队列(memory read and write queue,MRWQ),在 Tuple 写入 MRWQ 后返回数据处理成功信号. 即通过内存读写将针对磁盘的 IO 读写延迟到拓扑处理之外,以提高持久化的性能和流式计算引擎的响应速度. (2)设置独立线程程序 Processor 来负责将 MRWQ 中的数据持久化. Processor 会在以下两种情况下将 MRWQ 中的数据批量持久化:(*a*)自动刷新时间到达(*b*)队列长度达到阈值. 虽然在系统中添加缓存队列会增加开销,但只在负责持久化的拓扑末端添加缓存队列,其资源消耗率相对整个流计算系统而言是有限的,并且可以较好的优化系统响应率.

算法 3 延迟持久化队列

输入:当前时间窗口 *CTW*,待持久化数据 *result*,持久化队列 MRWQ 长度 *length*,MRWQ 的队列长度阈值 *maxLength*,当前时间 *currentTime*
输出:无

```
1.  BEGIN
2.    FOR CTW DO
3.      Get length of MRWQ;
4.      Get currentTime of System;
          //If the length of MRWQ does not exceed the threshold maxLength
5.      IF length < maxLength THEN
6.        Put result into MRWQ;
7.      ELSE//If the length of the MRWQ exceeds the threshold maxLength
8.        Persistence MRWQ;
9.        lastTime = currentTime;//Record the current time
10.       Put result into MRWQ;
11.     ENF IF
          //When the time reaches a limit updateTime
12.     If currentTime-lastTime >= updateTime THEN
13.       Persistence MRWQ;
14.       lastTime = currentTime;//Record the current time
15.     ENF IF
16.   END FOR
17. END
```

5 实现

我们在 Storm 中实现了上节设计的性能优化方法. 首先将 Storm 与 Kafka③(包含 KafkaSpout 和 KafkaBolt)进行整合,并通过 Zookeeper 实现了 Storm 中 Task 间的消息通信队列,以保证 $Task_i$ 与其上游 C_i 进行信号传递. 图 6 给出了基于上述技术的 Task 通信模型,其核心流程如表 2 所示,该通信模型是优化方法实现的技术基础.

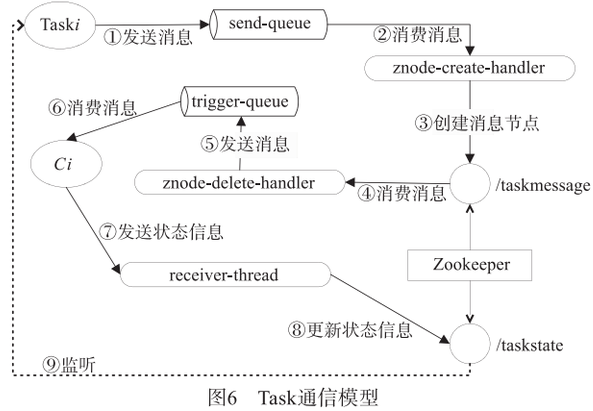


图6 Task通信模型

动态逐级反压:我们在 Storm 的处理组件 Bolt 中嵌入动态逐级反压算法,不同的 Bolt 会根据自身流量独立进行反压调整,Tasks 之间的反压信号通过 Zookeeper 消息队列通信,这种异步方式可能会导致下游 Tasks 的统计信息与上游 Tasks 统计信息有轻微不一致的情况,但是这种不一致并不影响最终计算的正确性. 动态逐级反

① Apache HBase:https://hbase.apache.org/
② Apache Cassandra:https://cassandra.apache.org/
③ Apache-Kafka:http://kafka.apache.org/

压结构如图 7 所示,从整体结构上观察,反压步骤如下:

①节点 Spout/Bolt 向下游发送一个数据(Tuple);

②下游收到 Tuple 的节点当前处于过载状态;

③当前节点 Trigger 触发反压,通过 Zookeeper 向上游传递反压信号;

④Zookeeper 接收并持久化反压信号,同时接受反压监听线程的监听;

⑤反压监听线程获取反压信号并通知节点进行反压,同时在 Zookeeper 中删除该信号;

⑥收到反压信号的节点触发器 Trigger 被触发,调整发射数据的速率后修改反压状态 State.

表 2　Task 通信模型核心流程

步骤编号	流程描述
①	$Task_i$发送消息到 send-queue 队列
②	znode-create-handler 线程负责消费 send-queue 中的消息
③	znode-create-handler 根据所消费的消息内容在 Zookeeper 的/taskmessage 目录下创建节点
④	znode-delete-handler 线程监听/taskmessage 目录,并负责消费目录下的消息
⑤	znode-delete-handler 将消息发送至 trigger-queue,并删除/taskmessage 目录对应节点
⑥	上游 C_i订阅到 trigger-queue 消费消息,根据消息内容进行反压等操作
⑦	receive-thread 负责接收 C_i的状态信息
⑧	receive-thread 将 C_i的状态信息更新至 Zookeeper 的/task-state 目录
⑨	所有的 Task 均监听/taskstate 目录,当目录有变化时,相应的 Task 会收到通知并作相关操作

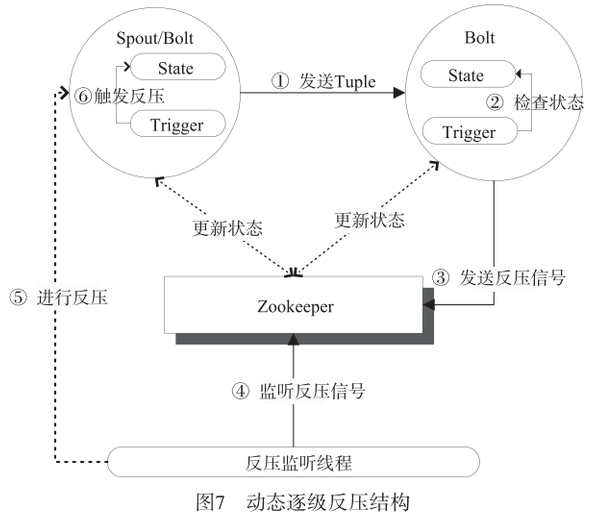


图7　动态逐级反压结构

无状态拓扑数据重放:无状态拓扑数据重放在 Storm 中的实现可依赖于 Storm 的记录级容错机制. 对于记录级容错,Storm 通过 Acker 确保每个 Tuple 在出错时被重发. 在此基础上,使用消息队列 Kafka 作为 Storm 的源队列,选择 Kafka 作为消息源队列的原因有两点:(1)Kafka 的快速、可持久化特性非常适合数据流的消息处理(2)Storm API 实现了 Kafka 与 Storm 间的无缝整合. 通过在 Storm 的确认机制中嵌入无状态拓扑数据重放,能将失败的 Tuple 重新写回 Kafka 队列. 由于无状态拓扑数据重放使用到的复制算法需要对 Kafka 的主题进行扩展,因此 KafkaSpout 被创建时,除了负责读取原主题(Eden 队列)中的数据外,它还负责在 Kafka 中创建相应拓扑的主题(From 队列与 To 队列),创建主题只会在拓扑首次运行时执行,通过 Storm 提供的 KafkaBolt 发送 Tuple 到 From 队列. 因此,当系统出现故障时,丢失的 Tuple 通过无状态拓扑数据重放可以进行恢复.

自适应拓扑替换:我们以 Tuple 完成延迟作为系统的数据计算延迟,通过检测 Tuple 完成延迟和 Tuple 失败数量来判断拓扑是否稳定. Storm 开放了 Thrift API,用来获取正在运行的拓扑性能和相关信息. 为了保证数据的有序性和依赖性,进行替换时新拓扑需在旧拓扑暂停后再开始执行. 替换过程中先使用 Storm 提供的 deactive 命令停止 Spout 的运行,系统会注销旧拓扑中的 Spout,使得旧拓扑中的 Tuple 超时,超时的 Tuple 因为无状态拓扑数据重放最终交由新拓扑来处理. 随后执行 *kill* 命令将所有旧拓扑的 Worker 关闭,并清除他们的状态,此时新拓扑已开始执行并能对数据流进行处理. 自适应拓扑替换机制可以分为两种情况,分别进行举例说明:

(1)当整个拓扑的负载较轻时,为节约系统资源,会降低拓扑的并发度,如图 8 为例,*T*2 从四个并发度缩容至两个并发度,*T*3 从两个并发度缩容至一个并发度.

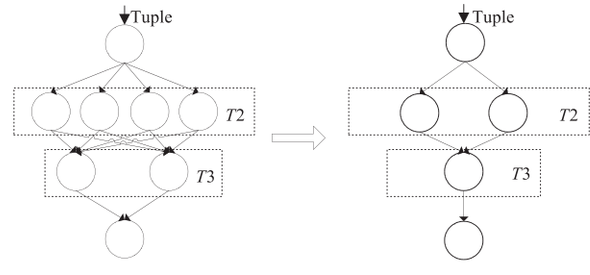


图8　拓扑缩容替换

(2)当整个拓扑过载时,将分配更多的资源给拓扑,如图 9 为例,*T*2 从二个并发度扩容到四个并发度.

延迟持久化队列:我们初步的实现是在持久层内部添加缓存队列 Inner-queue,如下图 10 所示,但随着流量增大,开始出现数据处理速度赶不上数据产生速度的现象,进而造成系统内存溢出. 另外,由于需要持久化的数

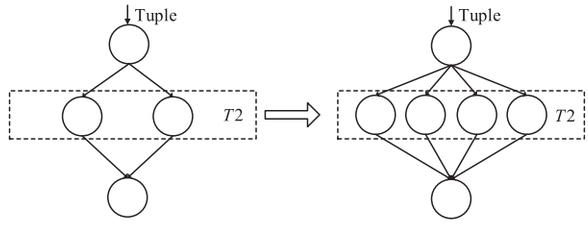


图9 拓扑扩容替换

据保存在内存中,若此时发生机器故障将会导致数据丢失且无法重放. 因此,最终方案中我们选择借助现有的开源分布式缓存框架 Memcached① 作为缓存队列的实现. 如图 11 所示,在持久层之上使用 Memcached 维护一个 MRWQ 队列,该队列在一个单独的持久化线程(Processor)中被批量刷新至持久层,进而充分利用分布式环境中机器的内存. 延迟持久化队列提供了两种条件以触发 Processor 与 Memcached 的交互及自动刷新:(1)自动刷新时间到达;(2)队列长度达到阈值.

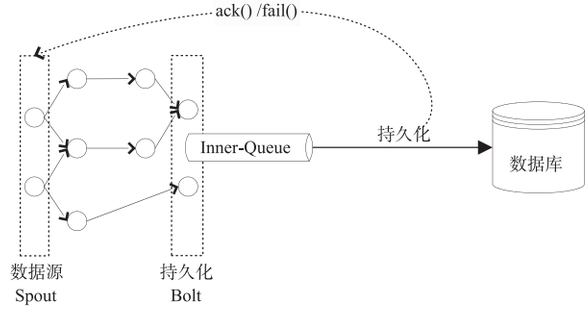


图10 内部缓存队列架构

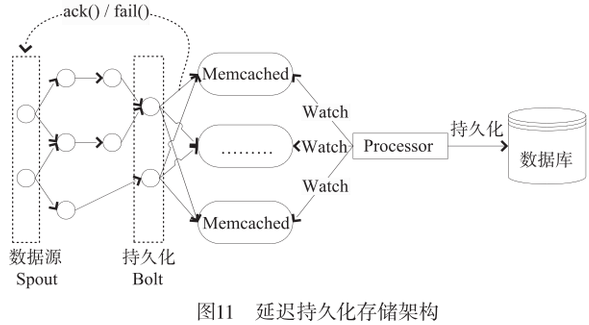


图11 延迟持久化存储架构

6 实验结果

基于上一步实现结果,我们将已优化的 Storm 安装在专为实验搭建的集群上. 整个实验集群由六台服务器节点组成,每个节点均有一个 2.2 GHz 的 Intel Core i7 CPU 的配置,服务器上运行的操作系统为 Ubuntu Linux 12.04,内存为 16G. 实验首先使用快速计数拓扑(Fast Word Count Topology)②和单词计数拓扑(Word Count Topology)③在吞吐量和数据计算延迟上对 Storm 优化实现与默认实现进行性能对比,以验证动态逐级反压与无状态拓扑数据重放的优化效果,在此基础上,我们再对延迟持久化队列和自适应拓扑替换方法进行仿真测试,以验证方法的有效性,为尽量避免误差,所有实验数据均是执行 50 次的平均值.

6.1 吞吐量测试结果及分析

本文使用快速计数拓扑测试系统吞吐量,如图 12 所示,该拓扑有一个 Spout 和两个 Bolt. Spout 重复地产生大小为 50 ~ 100byte 的随机字符串作为输入 Tuple,通过 shuffleGrouping 连接 SplitSentence,SplitSentence 发射从 Spout 接受的 Tuple,提取字符串中的单词. WordCount 接收 SplitSentence 发送的 Tuple,通过计数器每收到一次 Tuple 进行递增并输出计数值.

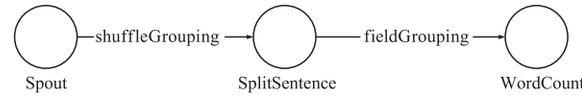


图12 吞吐量测试拓扑

通过设置不同的 Spout 和 Bolt 并发度来测试不同计算资源情况下拓扑的吞吐量. 参数设置如下表 3 所示.

表 3 并发度参数设置

组件\用例	用例 1	用例 2	用例 3	用例 4
Spout	1	1	1	1
SplitSentence	1	2	2	4
WordCount	1	1	2	6

图 13 显示了在不同测试用例上 L(本文优化后的实现)、D(Storm1.1.0 默认实现)和 H(Heron0.17.5 默认实现)的吞吐量随时间变化曲线. 如图 13(a)和图 13(b)所示,L 的吞吐量为 60323tuple/s 和 61307tuple/s,D 的吞吐量为 50836tuple/s 和 53614tuple/s,H 的吞吐量为 53036tuple/s 和 55762tuple/s,L 与 D 相比在吞吐量上约有 17% 左右的提升,与 H 相比在吞吐量上约有 11% 左右的提升. 随着并发度和机器资源的继续增加,Tuple 重放开始减少,D 与 H 的吞吐量逐渐接近 L 的吞吐量,图 13(c)中 L 的吞吐量为 64535tuple/s,D 与 H 的吞吐量分别为 60312tuple/s 和 62188tuple/s,L 与 D、H 相比提升了 3% ~ 7%,在图 13(d)例子中 L 的吞吐量为 67063tuple/s,D 与 L 基本持平,H 虽然在实验开始时获

① Memcached:https://memcached.org/

② Fast Word Count Topology, https://github.com/273539918/StormPerformanceTest/blob/master/FastWordCountTopology.java

③ Word Count Topology, https://github.com/273539918/StormPerformanceTest/blob/master/WordCountTopology.java

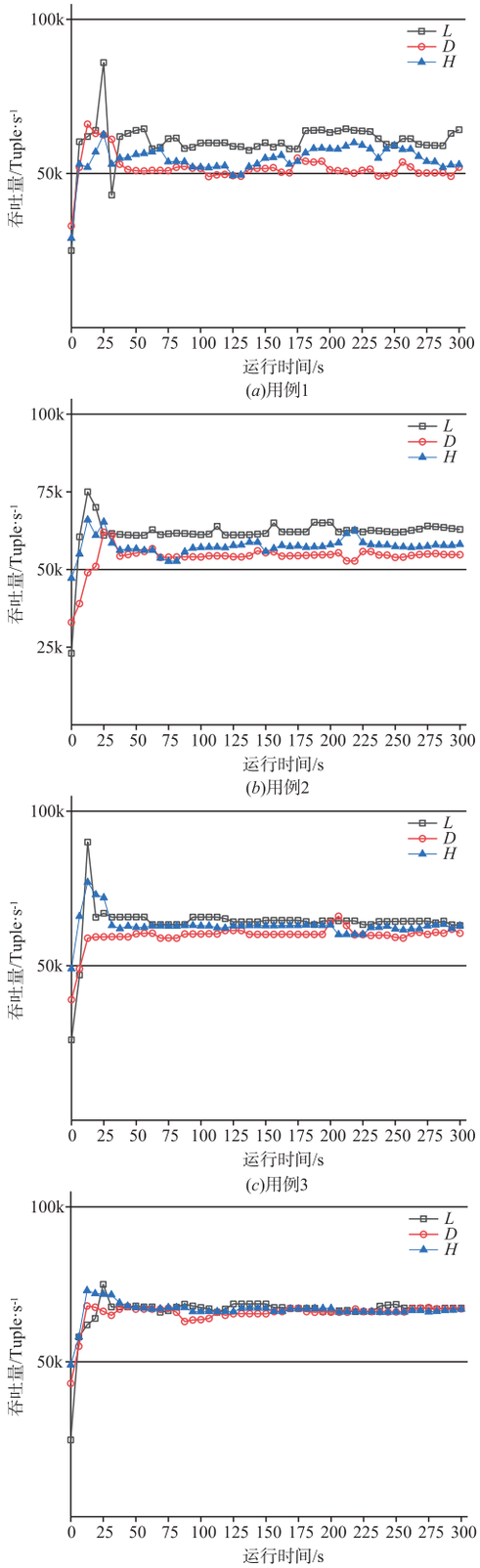


图13　吞吐量测试

得了较高的吞吐量,但在 50s 之后吞吐量也趋于稳定,与 L、D 基本持平. 因此,从实验结果来看,在最优情况下,*L* 能提升 3% ~17% 的吞吐量,最差情况下 *L* 与 *D*、*H* 的性能接近相等.

6.2　延迟测试

Word Count 是一个著名的 MapReduce 应用程序,本文设计了一个流计算版本的 Word Count 用于比较性能. 拓扑结构如图 14,输入使用爱丽丝梦游仙境英文版[29]作为词文件,并存入 Kafka 队列供 Spout 读取,Spout 一次从 Kafka 中读取一行,在字段中添加时间戳,kafkaWordSplitter 通过 shuffleGrouping 连接到 Spout,该 Bolt 将每行拆分成单词,并使用 fieldGrouping 将 Tuple 发送给 wordCountBolt. 拓扑的最后一个阶段是 persistenceBolt,它负责持久化数据,此外,该阶段根据字段中的时间戳与当前时间的差值求出数据计算延迟. 拓扑的并行度设置如表 4.

表 4　处理延迟并行度设置

	Spout	Kafkaword Splitter	Word Count Bolt	Persistence Bolt
并行度	1	2	4	1

图 15 显示了 *L*(本文优化后的实现)和 *D*(Storm1.1.0 默认实现)的数据计算延迟随时间变化曲线. *L* 与 *D* 相比会推迟 10 秒运行,主要原因是无状态拓扑数据重放和自适应拓扑替换需要创建 Kafka 的主题(Eden、From 与 To 队列),因此启动时间要比 *D* 慢. 在 20 秒左右,*D* 与 *L* 的延迟先后趋于稳定. 从多次实验平均结果来看,平均延迟基本处于稳定值(此时,*D* 为 485ms,*L* 为 380ms),偶尔有一些处理延迟比较大(图中尖刺),分析认为这是 Spout 线程偶尔获得较大的资源,导致了发送能力的增大,平均情况下 *L* 比 *D* 的 Tuple 延迟要低 20%.

6.3　延迟持久化队列测试

本文在 Word Count 实验(6.2 节)基础上进行延迟持久化队列测试. 首先人为控制让 persistenceBolt 的线程休眠一段时间,这将使 persistenceBolt 逐渐成为拓扑的性能瓶颈. Spout 一次从 Kafka 中读取一行,在字段中添加时间戳,在 Tuple 进行持久化后取出该时间戳,并获取数据计算延迟. 图 16 显示了 *L*(延迟持久化队列)和 *D*(直接存储)的平均数据计算延迟随时间变化曲线. *D* 由于 persistenceBolt 的延迟导致了拓扑整体的延迟增加,Tuple 的延迟呈持续上升状态. *L* 因为延迟持久化队列的存在,基本不受持久层性能的影响,稳定在 380ms. 测试结果表示,*L* 通过分布式缓存队列将针对磁盘的 IO 读写延迟到拓扑处理之外,极大地提高持久层的性能.

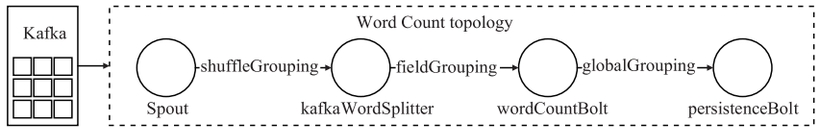


图14 处理延迟测试拓扑

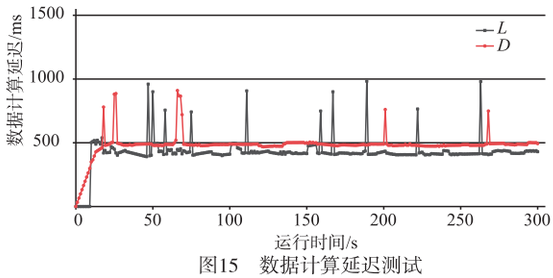


图15 数据计算延迟测试

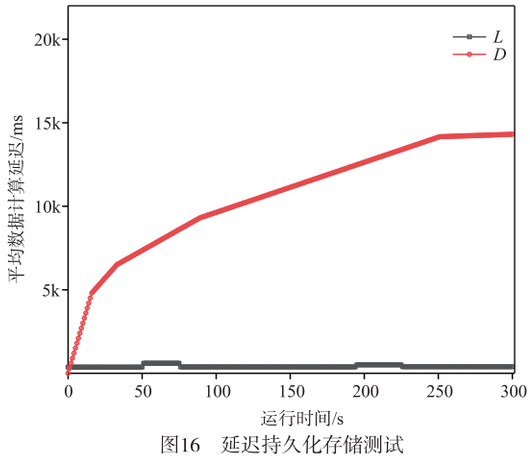


图16 延迟持久化存储测试

6.4 自适应拓扑替换测试

本节实验基于 Word Count(6.2 节)的基础上进行. 将自适应拓扑替换与 Storm 的 rebalance 方式进行对比,以拓扑扩容为例,扩容时它们都会为拓扑分配更多的资源. 仿真拓扑过载和替换过程,我们将 WordCount-Topology 的 kafkaWordSplitter 和 wordCountBolt 人为控制延迟若干毫秒,并将并行度设置为 1,使得整体拓扑处于过载状态. 图 17 显示了 L(自适应拓扑替换)和 D(Storm 默认 rebalance 方式)的平均数据计算延迟随时间变化曲线. 从图 17 可以看到 L 在大约 40s 开始使用新的拓扑来替换,由于旧拓扑停止时,新拓扑已启动,因此系统不需要暂停. 虽然延迟之后仍在上升,但趋势变缓. L 在 80s 再次进行扩容. 随着资源增多,延迟上升的趋势再次变缓. 由于 L 每次进行替换前新拓扑需等待旧拓扑中的数据处理完,并且在这个过程中 Spout 不发送数据,因此延迟会有突降到 0 的情况(即拓扑中没有计算数据). 对于 D,分别需要在 40s 和 90s 手动调用 rebalance 命令,可以看到每次进行 rebalance 时系统都有 22s 左右的时间不可用,这对于实时系统而言会造成

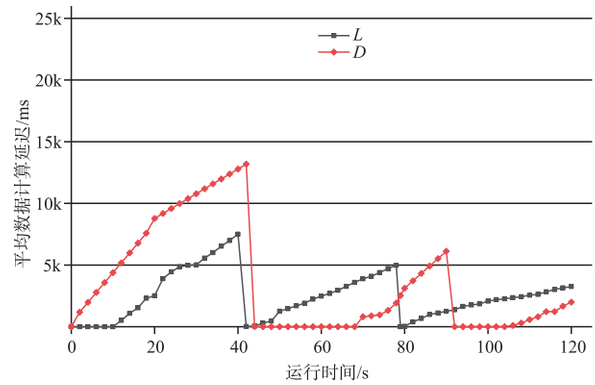


图17 自适应拓扑替换测试

极大影响. 实验结果表明,L 与 D 相比:(1)扩容过程对用户透明;(2)扩容过程系统无需暂停.

7 结束语

本文从大数据流式计算的特征切入,以优化大数据流式计算为核心,提出了基于拓扑优化的方法来改善 Storm 的性能,实验证明,本文的实现与 Storm 默认实现相比:(1)能有效提高系统吞吐量,最优情况下提高 17%,最差情况下与 Storm 默认实现接近相等.(2)能有效改善数据计算延迟,最优情况下提高 20% 的处理速度.(3)将针对磁盘的 IO 读写延迟到拓扑处理之外,极大地提高了持久层存储的性能和流计算系统的处理速度.(4)自适应地进行资源动态调整,系统无需暂停.

参考文献

[1] Shieh CK, Huang SW, Sun LD, et al. A topology-based scaling mechanism for Apache Storm [J]. International Journal of Network Management, 2016, 27(3): 63 – 68.

[2] Dean J, Ghemawat S. MapReduce: A flexible data processing tool[J]. Communications of the ACM, 2010, 53(1): 72 – 77.

[3] 孙大为,张广艳,郑纬民. 大数据流式计算:关键技术及系统实例[J]. 软件学报,2014,25(4):839 – 862.

[4] Akidau T, Balikov A, Bekiro, et al. MillWheel: fault-tolerant stream processing at internet scale[J]. Proceedings of the Vldb Endowment, 2013, 6(11): 1033 – 1044.

[5] Gulisano V, Jimenez-Peris R, Patino-Martinez M, et al. StreamCloud: An elastic and scalable data streaming system[J]. IEEE Transactions on Parallel & Distributed Sys-

tems,2012,23(12):2351 – 2365.
[6] Shkapsky A,Yang M,Interlandi M,et al. Big data analytics with datalog queries on spark[A]. Proceedings of the 2016 International Conference on Management of Data (SIGMOD2016)[C]. USA:2016. 1135 – 1149.
[7] Nair LR,Shetty SD,Shetty SD. Applying spark based machine learning model on streaming big data for health status prediction[J]. Computers & Electrical Engineering,2018,65(1):393 – 399.
[8] Barber R,Garcia-Arellano C,Grosman R,et al. Evolving databases for new-gen big data applications[A]. The Eighth Biennial Conference on Innovative Data Systems Research(CIDR 2017)[C]. USA:2017.
[9] 欧阳建权,周勇,唐欢容. 基于 Storm 的在线序列极限学习机的气象预测模型[J]. 计算机研究与发展,2017,54(8):1736 – 1743.
[10] Heinze T,Aniello L,Querzoni L,et al. Tutorial:Cloud-based data stream processing[A]. Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems[C]. Mumbai,India:2014. 238 – 245.
[11] Iqbal MH,Soomro TR. Big data analysis:apache storm perspective[J]. International Journal of Computer Trends & Technology,2015,19(1):9 – 14.
[12] Xhafa F,Naranjo V,Caballé S. Processing and analytics of big data streams with yahoo! S4[A]. 2015 IEEE 29th International Conference on Advanced Information Networking and Applications(AINA-2015)[C]. South Korea:2015. 263 – 270.
[13] Borthakur D,Gray J,Sarma JS,et al. Apache hadoop goes realtime at Facebook[A]. Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data(SIGMOD 2011)[C]. Greece:2011. 1071 – 1080.
[14] Qian Z,He Y,Su C,et al. TimeStream:reliable stream computation in the cloud[A]. Proceedings of the 8th ACM European Conference on Computer Systems(EuroSys 2013)[C]. Czech Republic:2013. 1 – 14.
[15] Dean J,Barroso LA. The tail at scale[J]. Communications of the ACM,2013,56(2):74 – 80.
[16] Li j,Sharma NK,Ports DR,et al. Tales of the tail:Hardware,OS,and application-level sources of tail latency[A]. Proceedings of the ACM Symposium on Cloud Computing(SoCC 2014)[C]. USA:2014. 1 – 14.
[17] Gulisano V,Nikolakopoulos Y,Papatriantafilou M,et al. Scalejoin:A deterministic,disjoint-parallel and skew-resilient stream join[A]. 2015 IEEE International Conference on Big Data (Big Data 2015)[C]. USA:2015. 144 – 153.
[18] Roy P,Teubner J,Gemulla R. Low-latency handshake join[J]. Proceedings of the Vldb Endowment,2014,7(9):709 – 720.
[19] Du G,Gupta I. New techniques to curtail the tail latency in stream processing systems[A]. Proceedings of the 4th Workshop on Distributed Cloud Computing(DCC 2016)[C]. Chicago,Ulinois:ACM,2016. 7.
[20] Dawei Sun,Shang Gao,Xunyun Liu,et al. State and runtime-aware scheduling in elastic stream computing systems[J]. Future Generation Computer Systems,2019,97:194 – 209.
[21] Guang Sun,Yingjie Song,Ziqin Gong,et al. Survey on streaming data computing system[A]. ACMTURC Conference on Artificial Intelligence and Security (ACM TUR-C)[C]. China:ACM,2019. 159:1 – 159:8.
[22] Georgios Chantzialexiou,André Luckow,Shantenu Jha. Pilot-Streaming:A stream processing framework for high-performance computing[A]. 2018 IEEE 14th International Conference on e-Science (e-Science)[C]. Netherlands:IEEE,2018. 177 – 188.
[23] 鲁亮,于炯,卞琛,等. 大数据流式计算框架 Storm 的任务迁移策略[J]. 计算机研究与发展,2018,55(1):71 – 92.
[24] Xu J,Chen Z,Tang J,et al. T-Storm:Traffic-aware online scheduling in storm[A]. 2014 IEEE 34th International Conference on Distributed Computing Systems(ICDCS 2014)[C]. Spain:IEEE,2014. 535 – 544.
[25] Peng B,Hosseini M,Hong Z,et al. R-Storm:Resource-aware scheduling in storm[A]. Proceedings of the 16th Annual MIDDLEWARE Conference (MIDDLEWARE 2015)[C]. USA:ACM,2015. 149 – 161.
[26] Li T,Tang J,Xu J. Performance modeling and predictive scheduling for distributed stream data processing[J]. IEEE Transactions on Big Data,2016,2(4):353 – 364.
[27] Kulkarni S,Bhagat N,Fu M,et al. Twitter heron:Stream processing at scale[A]. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD 2015)[C]. Australia:ACM,2015. 239 – 250.
[28] Goetz P T,O′Neill B. Storm Blueprints:Patterns for Distributed Realtime Computation[M]. U. K:Packt Publishing Ltd,2014.
[29] Carroll L. Alice′s Adventures in Wonderland[M]. Canada:Broadview Press,2011.

作者简介

陆佳炜　男,1981 年 9 月出生于浙江湖州,讲师,研究方向为大数据,云计算、服务计算.
E-mail:viivan@ zjut. edu. cn

吴 涵 男,1995 年 2 月出生于福建顺昌,硕士研究生,研究方向为大数据,云计算.
E-mail:wuhan@ zjut. edu. cn

肖 刚(通信作者) 男,1965 年 4 月出生于浙江上虞,教授,博士生导师,研究方向为云制造、智能制造.
E-mail:xg@ zjut. edu. cn