



面向数据流处理的元组跟踪方法

杜华明^{1,3}, 张 鹏^{2,3}, 徐克付^{2,3}, 谭建龙^{2,3}, 李 焱⁴

(1. 中国科学技术大学软件学院 合肥 230051;

2. 中国科学院信息工程研究所 北京 100093;

3. 信息内容安全技术国家工程实验室 北京 100093;

4. 国家计算机网络应急技术处理协调中心 北京 100029)

摘 要: 为了保证数据流中的每个元组得到可靠处理,传统的方法需要在内存中保存每个元组,直到它们被数据流处理系统正常处理,因此会带来很大的内存开销。为此提出了一种既能够保证元组得到可靠处理,又能够节省内存开销的元组跟踪方法。该方法包括内存分配策略、元组跟踪单元选择策略和校验值更新策略,这3个策略使得元组跟踪单元只保留元组标识符的异或校验值而不是元组减少内存开销,同时通过改进一致性散列变换实现元组跟踪单元的负载均衡。内存开销和负载均衡的相关实验表明,该方法能够有效实现对元组的跟踪和可靠处理。

关键词: 数据流处理;可靠性;数据流;负载均衡

doi: 10.3969/j.issn.1000-0801.2013.10.010

TTDSP: A Cost-Effective Approach to Tracking Tuple in Data Stream Processing

Du Huaming^{1,3}, Zhang Peng^{2,3}, Xu Kefu^{2,3}, Tan Jianlong^{2,3}, Li Yan⁴

(1. School of Software Engineering, University of Science and Technology of China, Hefei 230051, China;

2. Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China;

3. National Engineering Laboratory for Information Security Technologies, Beijing 100093, China;

4. National Computer Network Emergency Response and Coordination Center, Beijing 100029, China)

Abstract: The traditional data stream processing systems will keep all the tuples in the memory until they have been processed in order to provide reliable tuple processing. Unfortunately, the strategy will take up much memory. To address this issue, a cost-effective approach to tracking tuples-TTDSP was proposed. The approach includes three strategies, namely memory allocation strategy, tuple acker selection strategy and checksum updating strategy, which make tuple acker to keep only the XOR checksum not the tuple in memory. Moreover, the tuple acker are load balancing through the improved consistent Hash. The experiments on memory overhead and load balancing show that this approach is able to track and process tuples effectively and reliably.

Key words: data stream processing, reliability, data stream, load balancing

* 国家高技术研究发展计划(“863”计划)基金资助项目(No.2011AA010705),国家自然科学基金资助项目(No.61003295),中国科学院战略性先导专项基金资助项目(No.XDA06030602)



1 引言

随着云计算、物联网等技术的兴起,数据正以前所未有的速度不断增长和积累,大数据时代已经到来,其中典型的3个特点就是:规模性、多样性和高速性^[1]。同时,大数据主要的处理模式包括批处理和流处理两种^[2]。批处理是先存储后处理,而流处理则是直接处理。流处理的基本理念是数据的价值会随着时间的流逝而不断减少,因此尽可能快地对最新的数据做出处理并且给出结果是所有流处理的共同目标。流处理系统从处理模型上可以分为集中式、分布式以及并行分布式。然而,无论哪种处理模型,提高流处理的可靠性都是其中的热点和难点。故障容错作为提高系统可靠性的一个方面已经被广泛研究,并且出现了很多成熟的技术。总的来说,一个故障容错协议必须包含两个部分:一是节点的故障检测和替换;二是故障节点丢失状态的恢复。其中,检测和替换是被动行为,而丢失状态的恢复则是一种需要持续保存可能丢失状态信息的主动行为。在参考文献[3]中,给出了一个查询算子的故障容错的3种技术:主动备份技术、被动备份技术和上游备份技术。这3种技术的主要区别在于出现故障时保存可能丢失状态信息的方式(也就是查询算子的状态)。

主动备份技术(或者主动复制)通过一个算子的备份节点提供算子的故障容错,其中这个备份节点中的算子处理与主节点的算子相同的元组。也就是说,当主节点的算子出现故障时,可以使用这个备份节点的算子替换它。这种容错技术会带来较高的开销,其中主要的开销是保存副本的空间开销,因为它们在数据处理的大部分时间中并没有被利用。此外,元组必须发送到多个节点,这也会带来额外的时间开销。最后,备份节点的算子必须和主节点的算子保持相同的元组处理顺序,这会产生额外的时间开销。此外,当主节点出现故障时,主动备份技术需要把主节点的输出流切换到备份节点,因此故障恢复的时间也较长。

被动备份技术把属于要备份节点的算子状态周期性地复制到备份节点上。复制可以不断地在备份节点上或者在专用的节点上进行,当主节点出现故障时,这些备份被安装到替换的备份节点上。一个算子的周期性的复制被称为校验。与主动备份相比,周期性的校验减少了主节点和备份节点之间需要发送的元组个数,所以被动备份的时间开销较少。另一方面,由于在最后一个校验点和出现故障

的这段时间内发送到主节点的所有元组都没有在备份节点中被维护,所以这些元组需要重新发送到备份节点上,因此导致被动备份的故障恢复时间较长。

上游备份是一种不同的备份机制,它不需要使用任何备份节点,只依赖上游节点和下游节点。上游节点定义了一个协议用于维护其所输出元组的状态,直到下游节点确认这些元组可以被删除。上游备份的核心思想是当主节点出现故障时,上游节点把所有在输出队列并且还没有被下游节点确认的元组重新发送到替换的节点上。上游备份的唯一开销就是维护上游节点发送的元组的空间开销。然而,由于主节点状态需要重建并且再次分别处理每个元组,因此上游备份的故障恢复时间会较长(恢复时间取决于主节点状态的恢复时间)。

对于数据流处理系统,故障容错仍然是其中的重点和难点。在工业界中,S4和Storm是当前流行的数据流处理系统,其中S4使用Zookeeper来协同集群中的任务分配,集群中活跃(active)节点被分配具体的任务,而空闲(idle)节点放在池中以在故障容错或者负载均衡时使用。特别地,一个空闲节点可以注册成为分配不同任务的多个活跃节点的备份节点。对于S4在运行一段时间后可能出现失败、基础设施更新、调度重新分配和应用更新等情况,参考文献[4]中提出了相应的策略:高可用策略、基于检查点的状态恢复策略和低时延的处理策略。对于Storm,参考文献[5]中介绍的容错技术仅是将元组不停地重发,以保证每个元组至少得到一次完整的处理。

在学术界中,参考文献[6]认为现有的故障容错技术(无论被动备份还是主动备份)都会在运行时增加一些时间开销。该文的一个主要工作是提出了一个预测模型,该模型会在输入元组处理失败时启动故障容错机制。这个模型需要分流器来持续监控一个节点状态,并且把它们标记为正常(normal)、警告(alert)和失败(failure)。当一个节点的状态从正常变为警告时,分流器会启动故障容错机制。为了构建分类器,这个模型定义了一段训练时间以预测未来可能出现的故障。当预测模型发现一个可能在短期内发生的故障时,它首先把故障节点的算子迁移到一个专用的节点上以减少故障和算子替换所造成的影响。当节点状态是警告时,为了收集用来预测相同类型算子未来出现故障的信息,可能出现故障的算子的监控力度会增加。然而,该工作的不足在于实际中能够被预测的故障类型是很少的。不仅如此,该文提出的故障容错技术只有在预测模型所需

的计算资源没有超过其他故障容错技术所需要的计算资源时才有效。

在参考文献[7]和参考文献[8]中,关注非确定性算子的故障容错。在数据流中,算子执行的非确定性可以通过对输入流元组到达次序敏感的函数或者依赖时间的函数来定义。如作者所述,用来提供这种算子故障容错的维护信息包括到元组到达次序信息和依赖元组到达次序敏感的函数信息。该文作者也研究了如何有效维护副本,使其能够克服严格同步副本中输入元组的次序所带来的限制,同时作者还研究了如何以多线程方式运行副本。

在参考文献[9]中,作者提出了一种混合主动备份和被动备份的容错协议,其中的核心思想是周期性地校验查询算子的状态,这些状态并不存放到专用的节点,而是存放到这个算子的空闲副本上。也就是说,算子的一个副本正在被其他节点所维护,这个副本不用接收主节点所处理的相同元组,它的状态只需要通过增量的校验点来持续地更新。当主节点出现临时故障时,算子的副本被启用并且开始处理和主节点所处理的相同元组。如果主节点出现永久故障时,那么这个查询会得到指示,开始只向算子的副本发送元组并且开始使用它的输出元组。

参考文献[10]在前期工作^[11]的基础上提出了一种基于异步校验机制(类似于模糊校验)的容错协议,该协议没有对算子状态和它的输出队列进行校验,而是让所有输出元组都包含窗口校验元组(专门用来描述窗口中间状态的元组),并且只有输出的数据流被持久化。当节点出现故障时,通过读取该节点的算子输出队列来查找最近的窗口校验信息,以计算出从哪个位置重新发送数据流的元组。如作者讨论的一样,由于这种校验方式在重建故障节点的算子状态时减少了重新发送元组的数量,所以可以减少故障恢复的时间。

StreamCloud^[12]在参考文献[10]的基础上对故障容错以下几个方面改进:

- 出现故障时重新发送元组的时间点的信息 `earliest timestamp` 只包含在输出元组的头部,减少了存储开销;
- `earliest timestamp` 是在线维护的,因此避免了在并行文件系统中查找数据的不必要的读取操作;
- 数据流持久化中通过采用一种对持久化信息自识别的命名方式避免了元数据的维护^[9],减少了故障容错过程对运行时的影响;
- 基于 `earliest timestamp` 的容错协议可以对重新部署

期间发生的故障进行容错。

然而,上述技术如果把故障粒度定为数据流中的每个元组,当元组数量很多时,跟踪这些元组是否已经被处理的内存开销会很大。因此需要一种既能节约内存又能够保证需要处理的每个元组都被处理的可靠方案。为此本文提出了一种既能够保证元组得到可靠处理又能够节省内存开销的元组跟踪方法。该方法包括内存分配策略、元组跟踪单元选择策略和校验值更新策略,这3个策略通过只保留元组标识符的异或校验值而不是元组来减少内存开销,同时通过改进一致性散列来实现元组跟踪单元的负载均衡。

2 元组跟踪器

流是由具有相同数据模式且无界限的元组序列组成的。与传统的数据库系统不同,数据流处理系统处理的数据是没有经过持久化的流数据,当且仅当满足查询条件时,数据流处理系统才会将查询结果返回给用户,所以该查询又可以称作连续查询。一个查询可以定义为一个有向无环图,并且图中每个节点都是一个操作算子,图中每条边可以表示的是数据流向。本文把查询的起点称为元组生成器(`spring`),处理单元称为元组处理器(`processor`)。元组生成器可以产生并发送数据流,其中的数据流中的元组被称为根元组。根元组的状态分为正在处理状态(`pending`)、处理失败状态(`failure`)和处理成功状态(`finished`)3种。元组处理器接收数据流并且处理数据流,处理后的数据流也可以发送给其他的元组处理器。元组生成器和元组处理器在元组的处理过程中会启动多个任务线程来并行处理元组。

元组生成器处理根元组后会产生多个元组,这些元组经过元组处理器处理后可能继续产生新的元组,直到元组处理器不再产生新元组为止,所有元组所形成的一个树状结构被称为元组树。元组树的根元组用 `springId` 唯一标识,其他元组用 `tupleId` 唯一标识。在元组树形成的过程中,元组生成器和元组处理器会不断地向元组跟踪器发送消息,元组跟踪器根据发送来的消息构造跟踪记录,并将跟踪记录存储在元组跟踪单元中(元组跟踪单元的选择策略在第2.2节介绍)。元组跟踪单元(`acker`)是一个跟踪元组处理过程的进程。跟踪记录是一个三元组,表示为 `<springId, taskId, checkValue>`,其中 `checkValue` 表示根元组的校验值,通过它判断根元组是否得到一次完整处



理。根元组的一次完整处理指的是元组树中的所有元组都被成功处理,如果元组树中的任意一个元组在指定的时间内没有被成功处理,那么元组生成器发出的根元组被认为处理失败。每个元组在处理的过程中通过向元组跟踪器发送元组的相关信息来对其跟踪记录的校验值进行更新。元组跟踪器的生命周期可分为启动、运行和终止 3 个阶段,在这 3 个阶段内,元组跟踪器和数据流处理引擎交互过程如图 1 所示。

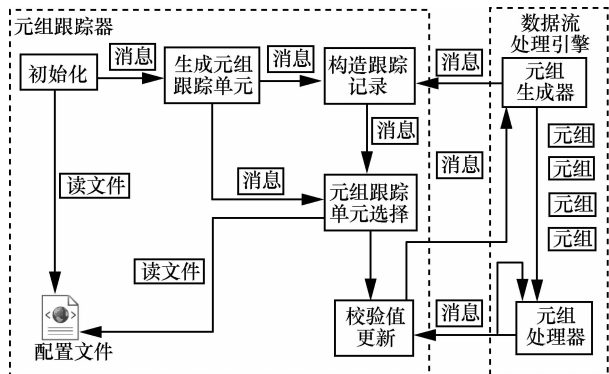


图 1 元组跟踪器的交互过程

在元组跟踪器启动阶段,首先是初始化元组跟踪单元的数量,元组跟踪单元的数量可以从配置文件中读取,也可以在元组跟踪器运行状态下,通过对外接口来修改元组跟踪单元数量。然后查看是否能够产生元组跟踪单元。如果能够正常产生元组跟踪单元,元组跟踪器开始接收元组生成器发送的 `springId` 和 `taskId`,以进入运行阶段。否则,元组跟踪器再次读取配置文件重新生成相应数量的元组跟踪单元。

元组跟踪器的运行阶段可以细划为内存分配、元组跟踪单元选择和校验值更新 3 个子阶段。如图 1 所示,当元组跟踪器接收到元组生成器发来的 `springId` 和 `taskId` 后,首先进入内存分配阶段,元组跟踪器为每个元组分配大约 20 byte 的内存空间来构造跟踪记录;然后进入元组跟踪单元选择阶段,通过元组跟踪单元选择策略将跟踪记录存储在不同的元组跟踪单元上;最后进入校验值更新阶段,元组生成器(元组处理器)会不断地向元组跟踪器发送消息(已处理的元组 ID 和新产生的元组 ID),元组跟踪器利用元组生成器(元组处理器)发来的消息不断地更新跟踪记录中的校验值,更新策略在第 3.3 节介绍,当校验值为 0 时,说明跟踪记录所跟踪的元组已经得到了完整的处理。此时,元组跟踪器将会通知元组生成器相应的任务线程,

任务线程将对应的元组的状态修改为已完成(finished)。最后,元组生成器会将状态为已完成的元组从内存中移除。当校验值不为 0 时,任务线程将对应的元组的状态修改为失败(failure),最后,元组生成器会将状态为失败的元组重新发送。

元组跟踪器和数据流处理引擎之间是松耦合关系,它可以独立于数据流处理引擎运行,当不需要元组跟踪器时,用户可以通过命令行来终止元组跟踪器所对应的进程。

3 元组跟踪策略

为了实现节省内存、负载均衡和可靠的元组处理,元组跟踪器采用的技术主要涉及内存分配、元组跟踪单元选择和校验值更新,下面具体介绍这 3 个策略。

3.1 内存分配策略

大数据具有规模大并且速度快的特点,要保证从元组生成器产生的每个根元组都得到至少一次完整的处理,则需要对每个根元组所形成的元组树中的每个元组进行跟踪,以确定元组树中的所有元组是否都得到完整的处理。但是,如果元组树中含有成千上万个节点,对元组树的跟踪所占用的内存会随着元组树中节点数的增加而呈现指数级增长,这样会导致内存溢出。针对跟踪元组树的内存开销问题,本文提出一种节约内存的方法,该方法只保留元组标识符的异或校验值而不是元组。其中,元组生成器接收根元组后,会向元组跟踪器发送 `springId` 和 `taskId`,然后,元组跟踪器利用 `springId`、`taskId` 以及 `checkValue` 构造跟踪记录。当这个根元组得到完整处理时,元组跟踪器会通知 `taskId` 对应的任务将根元组从内存中移除,否则,元组跟踪器会通知 `taskId` 对应的任务重新发送该根元组。

3.2 元组跟踪单元选择策略

由于元组生成器产生元组的数量多,如果仅使用一个元组跟踪单元来跟踪元组生成器产生的所有根元组,那么元组跟踪单元的负载会很高。因此,元组跟踪器需要使用多个元组跟踪单元来跟踪元组生成器产生的根元组。为了使各个元组跟踪单元跟踪元组的数量尽量均衡,需要一个将元组的跟踪记录分配到不同的元组跟踪单元的分配策略,使得各个元组跟踪单元负载相对均衡,这样不仅可以减小单个元组跟踪单元的负载压力,同时也可以提高整体性能。如果某个元组跟踪单元异常终止,元组跟踪器会将该元组跟踪单元的跟踪记录分配给其他的元组跟踪单元。

本文中,元组跟踪单元的选择采用了改进的一致性散列策略,其中的原理如下:元组跟踪器使用散列函数将元组生成器产生的根元组 ID 映射到环上的某一个值,环是一个由 $0 \sim (2^{32}-1)$ 的数值组成的空间。然后将一个元组跟踪单元及其副本分别映射到环上的某一个值,每个元组跟踪单元跟踪逆时针方向上与其距离最近的元组,这样每个元组跟踪单元所跟踪的元组数量就会相对均衡。具体的分配步骤如下。

首先将元组映射到一个 32 bit 的 key 值,该映射首先初始化全局变量 $hash=0$,然后将字符串中每个字符从右到左顺序执行如式(1)所示的计算式:

$$hash=hash \times 131 + \text{字符的 ASCII} \quad (1)$$

最后执行如式(2)所示的计算式:

$$key=hash/(2^{32}-1) \quad (2)$$

所得的 key 值对应环中的某个值。

例如根元组、根元组 2...根元组 6 和 acker A、acker B,将这 6 个根元组和两个 acker 的 ID 映射到环上,散列函数为 $key=hash(value)$,该函数封装的是式(1)和式(2)的逻辑,value 是 springId 或 ackerId,然后将映射的结果封装成 $location=<key, type, active>$ 。其中 type 为 0,表示根元组的映射;type 为 1,表示 acker 的映射。active 默认值为 0,表示 acker 未启动;active 为 1,表示 acker 正常运行;active 为 2,表示 acker 异常终止。映射的结果在环中的分布如图 2(a)所示。

对于环中的每个根元组,从根元组的 key 值出发,沿顺时针方向旋转搜索,当遇到第一个 acker 时,将元组的 location 存储在该 acker 上,因为 springId 和 ackerId 的散列值是固定的,因此这个元组和 acker 的关系必然是唯一和确定的。理想的散列结果是将所有元组均匀分配到各 acker 中,采取的策略就是将一个 ackerId 映射到两(N)个位置,这样可以保证跟踪记录相对均匀地分配到各个 acker 中。此时 value 为 acker_id#1, acker_id#2, ..., acker_id#N, hash 值 $key=hash(value)$, hash 值在环上的分布以及分配结果如图 2(b)所示。

在元组跟踪器运行过程中,由于某些原因导致 acker 的数量减少,根据图 2(a)和图 2(b)所描述的映射方法,这时受影响的将仅是沿 acker B₁ 和 acker B₂ 逆时针遍历直到下一个 acker(acker A₂ 和 acker A₁)之前的 location,也是本来映射到 acker B 上的那些 location。那么,仅需要将根元组 2 对应的 location 分配给 acker A₂,根元组 4 和根元组 6

对应的 location 分配给 acker A₁ 即可,元组的重新分配如图 2(c)所示。

假如因为元组的数目过多(仅有 acker A 和 acker B 记录这些元组对应的 location 状态负载过大)而增加 acker 或者通过修改元组跟踪器的 acker 数量 N 值增加 acker(假设新增 acker C)。通过图 2(b)中所提到的将 ackerId 映射到环的方法,acker C 的两个 hash 值分别映射到根元组 3 和根元组 5、根元组 4 和根元组 6 所对应的 location 之间,这时受影响的元组将仅是那些沿 acker C₁ 或 acker C₂ 逆时针遍历直到下一个 acker B₂ 和 acker A₂ 之间的 location(它们本来也是映射到 acker B₁ 和 acker A₂ 上的),将这些 location 重新映射到 acker C₁ 和 acker C₂ 上即可,映射的结果如图 2(d)所示。

以上所描述的就是元组跟踪单元选择策略的全过程,通过使用改进的一致性散列将一个 ackerId 映射到环上的多个位置,不仅能够保证元组的跟踪记录分配的相对均衡,同时也保证增加或者减少元组跟踪单元只会影响小部分已经分配的跟踪记录。

3.3 校验值更新策略

为了保证元组生成器产生的根元组至少得到一次完整的处理,当元组没有被成功处理时,元组跟踪器将会重发没有得到成功处理的根元组。元组跟踪器通过跟踪记录中的校验值(0/非 0)来判断元组生成器产生的根元组是否得到完整的处理,如果某个元组没有得到完整处理(校验值为非 0),那么元组跟踪器会通知元组生成器中处理该元组的任务(task)重新发送该元组。

跟踪记录通过对校验值判断根元组是否已经得到完整的处理。不管这棵元组树多大,它只是简单地把这棵树上的所有已处理的 tupleId(根元组 ID 除外)和新产生的 tupleId 进行异或(XOR)运算,并以此结果更新校验值。当校验值为 0 时,表示跟踪记录中对应的根元组被完整地处理,因为元组树中的每个 tupleId 都出现两次,所以异或的结果为 0,由此证明根元组产生的元组树中的所有元组都得到完整处理。反之,没有得到完整处理,这是因为如果在处理元组的过程中出现元组丢失,那么元组树中每个 tupleId 不会出现两次,所以结果不为 0,此时通过跟踪记录中的 taskId 通知元组生成器中相应的任务重发 springId 对应的根元组。因为校验值是 64 bit 的,所以元组树存在未被处理的元组但是异或结果为 0 的概率是 1/264,因此可以忽略。下面描述一下 checkValue 更新的过程。

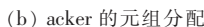
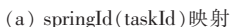


图 2 元组跟踪单元选择策略

以上过程就是某一个跟踪记录中校验值的变化过程。

下面以元组 springId 为 01001 为例,介绍它的跟踪记录校验值的更新的全过程。如图 3 所示,元组生成器产生 springId 为 01001 的根元组,元组生成器处理根元组并产生了 tupleId 为 01010 和 01011 两个元组,此时没有需要确认的元组,元组生成器将这两个新产生的元组 ID 发送给相应的元组跟踪单元,元组跟踪单元将这两个元组 tupleId 为 01010 和 01011 进行异或运算,将得到的结果与校验值做异或运算。计算所得的结果更新校验值。将 tupleId 为

01010 的元组被发送到元组处理器 1 中, 将 tupleId 为 01011 的元组被发送到元组处理器 2 中, 计算过程如图 3(a) 所示。

元组处理器 1 处理 01010 这个元组, 处理元组的结果是产生了新元组, 其 tupleId 为 01100。元组处理器 1 将已处理的元组 tupleId 和新产生的元组 tupleId 发送给相应的元组跟踪单元, 元组跟踪单元将 01010 和 01100 进行异或运算, 用所得的结果与校验值做异或运算, 用计算所得的结果更新校验值, 将 tupleId 为 01100 的元组传到元组处理器 3 中, 计算过程如图 3(b) 所示。

元组处理器 2 处理 01011 这个元组, 处理元组的结果是产生了新元组, 其 tupleId 为 01101。元组处理器 2 将已处理的元组 tupleId 和新产生的元组 tupleId 发送给相应的元组跟踪单元, 元组跟踪单元将 01011 和 01101 进行异或运算, 用所得的结果与校验值做异或运算, 用计算的结果更新校验值, 将 tupleId 为 01101 的元组发送到元组处理器 3 中, 计算过程如图 3(c) 所示。

元组处理器 3 处理 01100 和 01101 这两个元组, 不再

有新的元组生成。那么, 元组处理器 3 仅将已处理的元组 tupleId 发送给相应的元组跟踪单元, 元组跟踪单元将 01100 和 01101 做异或运算, 用所得的结果与校验值做异或运算, 用计算所得的结果更新校验值, 计算过程如图 3(d) 所示。

4 实验结果

下面通过实验验证元组跟踪器的内存开销和负载。本实验环境是采用 4 台 4 核主频 2.40 GHz 的 PC 机做服务器, 内存 4 GB。本实验所定义的查询仅有一个映射 (map) 算子和一个聚合 (aggregate) 算子。数据源是文件中读取的英文文本数据, 本实验的目标是计算出文本中不同的单词在文本中出现的次数。元组生成器将文本中的每一行数据封装为一个根元组, map 算子就是将每行的文本以空格为分隔符将根元组分割成若干元组 (单词), 并将每个元组 (单词) 发给 aggregate 算子做聚合计算。

4.1 实验一: 单个元组跟踪单元的内存开销

实验一验证仅有一个元组跟踪单元时, 输入元组的数量对元组跟踪单元占用内存的影响。输入的元组数从

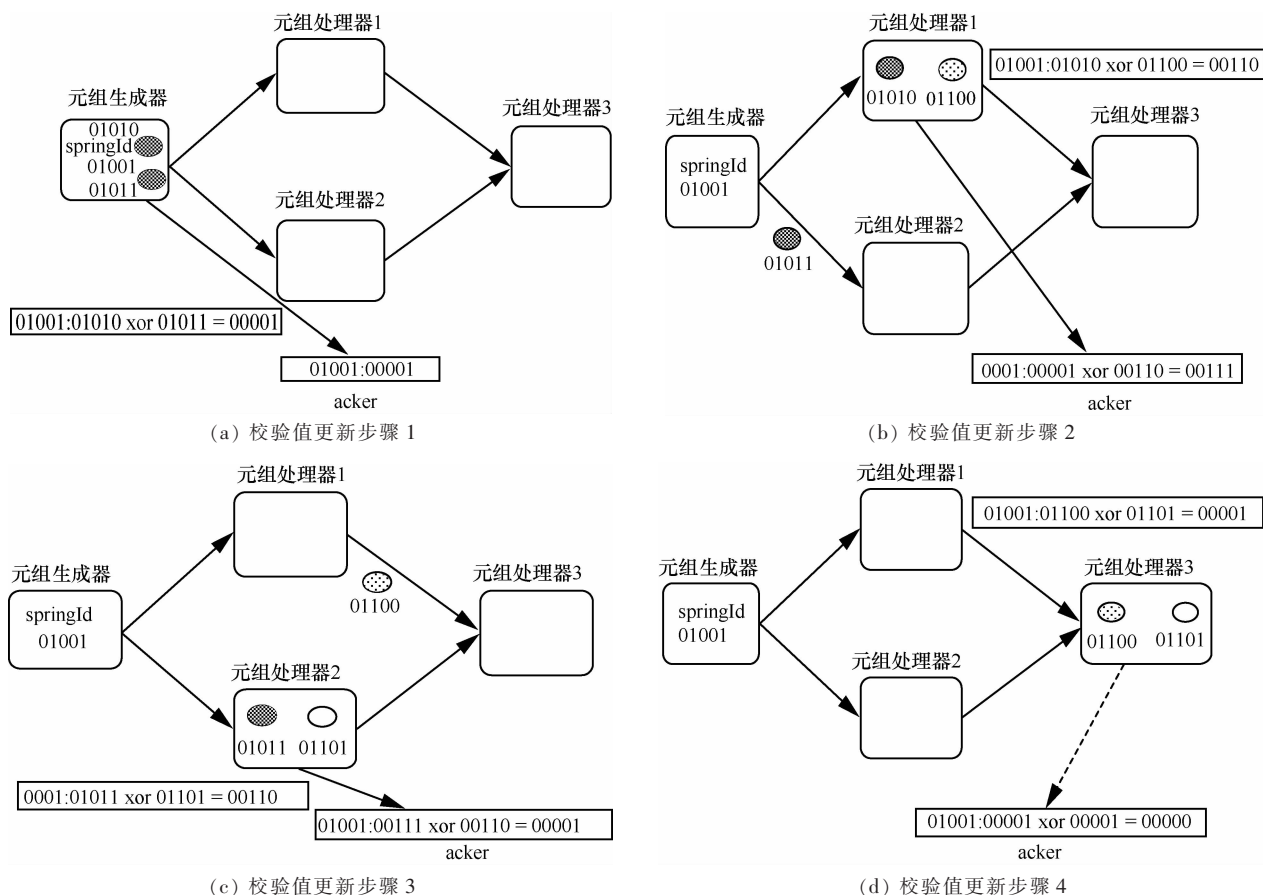


图 3 校验值更新流程



10 万到 90 万不等,元组跟踪单元的内存变化情况如图 4(a)所示,当输入的元组个数是 10 万时,上文已经介绍了一个跟踪记录仅消耗 20 byte 的内存,理论上 10 万应该消耗 1.9 MB 内存,而实际仅消耗 0.86 MB 内存,内存消耗减少了 55%左右。当输入元组个数是 20 万时,理论计算应该消耗 3.8 MB 左右,而实际上仅消耗 1.59 MB,内存消耗减少了 58%左右。出现以上现象是因为当元组跟踪单元确定根元组已经得到完整的处理后,元组跟踪器会通知元组生成器将对应根元组移除内存,与此同时元组跟踪单元也会将相对应的跟踪记录移除内存,也就是说在不断地构造新跟踪记录的同时也将已处理的跟踪记录移除内存,所以该阶段内存的消耗的增长率较小,从图 4(a)中可以看出元组的数量较少时,元组跟踪器的内存开销较小。当输入的元组增大到 40 万后,内存开销增长率会平缓地增大,由于程序对元组的跟踪能力有限,所以内存开销会增加。当输入元组增长到一定数量后,受机器内存的影响,元组跟踪器的内存开销将不会增加,内存开销增长率为 0。同时,从图中可以看到,随着输入元组数量的增多,无元组跟踪单元的数据流处理系统的内存开销线性增长,当输入元组个数为 90 万时,无元组跟踪单元对元组的跟踪需要消耗大约 69 MB 内存,而元组跟踪单元对元组的跟踪仅仅消耗大约 16 MB 的内存。所以图 4(a)也从侧面反映了另外

的一个问题:即使使用单个元组跟踪单元来跟踪元组,内存开销也是很小的。第 4.2 节观察在使用多个元组跟踪单元时,是否能够进一步地减少单台机器的内存开销。

4.2 实验二:多个元组跟踪单元的内存开销

实验二是观察随着元组跟踪单元的增加,各元组跟踪单元的内存开销(这里所说的内存开销指的是集群中各机器内存平均开销)情况。将输入元组限定在 50 万个,当元组跟踪单元的数量为 1 时,可以从图 4(b)中看出,内存开销略小于 7 MB,当元组跟踪单元增多时,元组跟踪单元内存开销不断减少,这是因为随着元组跟踪单元数量的增加,原来分配到一个元组跟踪单元的跟踪记录被多个跟踪单元均匀分配,所以随着元组跟踪单元的增加,内存开销逐渐减少。从图中可以看出,当元组跟踪单元的数量超过 6 个时,元组跟踪单元的内存开销减少幅度越来越小,这是被集群自身节点个数所限制,因为在集群中机器数量较少的情况下,随着元组跟踪单元的数量不断增多,集群中每台机器中运行的元组跟踪单元数量相对较多,那么内存开销会逐渐增大。

4.3 实验三:多个元组跟踪单元的负载

实验三主要是验证第 3.2 节的元组跟踪单元选择策略能否使得各个元组跟踪单元存储的跟踪记录相对均衡,要验证的是输入元组的数量对元组跟踪单元负载均衡的影

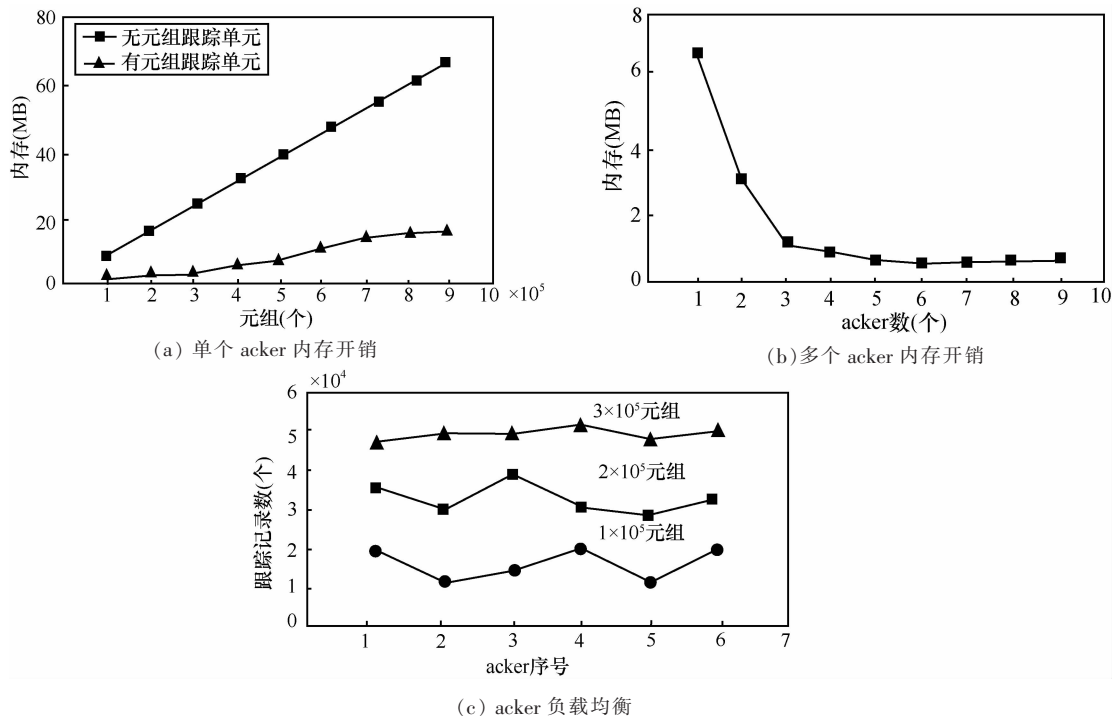


图 4 元组跟踪器的内存开销和负载

响,分别做 3 次实验,每次实验的元组跟踪单元的数量都设置为 6 个,并且每次实验输入元组数量分别为 10 万个、20 万个和 30 万个。实验结果如图 4(c)所示。从图中可以看出,随着输入元组数量的增多,曲线逐渐趋于横线,这也说明了输入的元组越多,各个元组跟踪单元存储的元组的总数量相对更均衡。这主要是因为元组跟踪单元选择策略是将各个元组跟踪单元及其副本的映射值均匀分布在环上,采用元组跟踪单元的副本策略可以使各个元组跟踪单元在环上的分布尽量均衡,这样输入元组的数量越多,将元组映射到环上的位置占用率就越高,根据第 3.2 节所述的跟踪记录分配方法,在数量上每个元组跟踪单元所存储的跟踪记录也就相对地均衡。

5 结束语

规模性、多样性和高速性是大数据处理必须要考虑的 3 个特征,很多研究工作围绕这 3 个特征正在展开。针对规模性和高速性,本文提出了一种面向数据流处理的元组跟踪方法,其中的最大特点是节约内存,该方法具体包括内存分配策略、元组跟踪单元选择策略和校验值更新策略,这 3 个策略通过只保留元组标识符的异或校验值而不是元组减少内存开销,同时通过改进一致性散列实现元组跟踪单元的负载均衡。内存开销和负载均衡的相关实验表明,该方法有效实现了元组的可靠处理。

参考文献

- 1 孟小峰,慈祥. 大数据管理:概念、技术与挑战. 计算机研究与发展,2013,50(1):146~169
- 2 Kumar R. Two computational paradigm for big data. <http://kdd2012.sigkdd.org/sites/images/summerschool/Ravi.Kumar.pdf>, 2012
- 3 Hwang J H, Balazinska M, Rasin A, *et al.* High-availability algorithms for distributed stream processing. Proceedings of 21st International Conference on Data Engineering (ICDE 2005), Tokyo, Japan, 2005
- 4 S4 distributed stream computing platform. http://incubator.apache.org/s4/doc/0.6.0/fault_tolerance/, 2013
- 5 Guaranteeing message processing. <https://github.com/nathanmarz/storm/wiki/Guaranteeing-message-processing>, 2013
- 6 Gu X H, Papadimitriou S, Yu P S, *et al.* Toward predictive failure management for distributed stream processing systems. Proceedings of 2008 the 28th International Conference on Distributed Computing Systems (ICDCS '08), Washington, DC, USA, 2008

- 7 Brito A, Fetzer C, Felber P. Minimizing latency in fault-tolerant distributed stream processing systems. Proceedings of 2009 the 29th IEEE International Conference on Distributed Computing Systems (ICDCS '09), Washington, DC, USA, 2009
- 8 Brito A, Fetzer C, Felber P. Multithreading-enabled active replication for event stream processing operators. Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems (SRDS '09), Niagara Falls, New York, USA, 2009
- 9 Zhang Z, Gu Y, Ye F, *et al.* A hybrid approach to high availability in stream processing systems. Proceedings of the 30th International Conference on Distributed Computing Systems (ICDCS '10), Washington, DC, USA, 2010
- 10 Sebeopou Z, Magoutis K. CEC: continuous eventual checkpointing for data stream processing operators. Proceedings of 2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN), Hong Kong, China, June 2011
- 11 Sebeopou Z, Magoutis K. Scalable storage support for data stream processing. Proceedings of the 26th Symposium on Mass Storage Systems and Technologies (MSST), Incline Village, Nevada, May 2010
- 12 Gulisano V, Jimenez-Peris R, Patino-Martinez M, *et al.* StreamCloud: an elastic and scalable data streaming system. IEEE Transactions on Parallel and Distributed Systems, 2012, 23(12):2351~2365

[作者简介]



杜华明,男,中国科学技术大学软件学院硕士研究生,主要研究方向为大数据处理。

张鹏,男,中国科学院信息工程研究所站博士后,主要研究方向为数据流处理与云计算。

徐克付,男,中国科学院信息工程研究所副研究员、博士生导师,主要研究方向为大数据与云计算、信息内容安全。

谭建龙,男,中国科学院信息工程研究所研究员、博士生导师,主要研究方向为云计算与网络安全。

李焱,男,国家计算机网络应急技术处理协调中心工程师,主要研究方向为云计算。

(收稿日期:2013-07-25)