流水行云:支持可扩展的并行分布式流处理系统

张 鹏[1,2],刘庆云[1,2],谭建龙[1,2],李 焱[3],杜华明[2]
(1.中国科学院信息工程研究所,北京 100093; 2.信息内容安全技术国家工程实验室,北京 100093;
3.国家计算机网络应急技术处理协调中心,北京 100029)

摘 要: 数据流处理系统,无论是集中式还是分布式,都需要克服单点瓶颈问题.不仅如此,如果数据流处理系统是静态配置的,那么还会出现处理节点供给不足或者过剩的情况,为此本文提出了一种支持可扩展的并行分布式数据流处理系统—流水行云,该系统根据有状态算子将查询拓扑划分为并行处理的子查询,并且通过有状态算子的分发器和收集器实现了数据流的保序,同时最大化减少并行处理的通信开销,不仅如此,结合负载均衡和重配置的可扩展技术使得该系统能够根据输入负载动态调整处理节点的负载和个数.60个节点组成的集群的实验证明了该系统的可扩展能力.

关键词: 流处理系统; 可扩展; 有状态算子; 负载均衡; 重配置
中图分类号: TN911.23 **文献标识码:** A **文章编号:** 0372-2112 (2015)04-0639-08
电子学报 URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2015.04.003

SPSPS:A Scalable Parallel-Distributed Stream Processing System

ZHANG Peng[1,2], LIU Qing-yun[1,2], TAN Jian-long[1,2], LI Yan[3], DU Hua-ming[2]
(1. *Institute of Information Engineering, Chinese Academy of Sciences, Beijing* 100093, *China*;
2. *National Engineering Laboratory for Information Security Technologies, Beijing* 100093, *China*;
3. *National Computer Network Emergency Response and Coordination Center, Beijing* 100029, *China*)

Abstract: The stream processing systems, whether centralized or distributed, have to overcome the single-node bottleneck. Moreover, their static configurations also make them either shortage or surplus of resources. To this end, this paper proposes a scalable parallel-distributed stream processing system named SPSPS. The system splits a query into parallel sub-queries according to stateful query operators to minimize the communication overhead in parallel processing, and achieves order-preserving tuple processing through the stateful operator's distributor and collector. Moreover, the scalability techniques with load balancing and reconfiguration support effective adjustment of resources depending on the incoming load. The experiments on a cluster with 60 nodes prove the scalability.

Key words: stream processing system; scalability; stateful operator; load balancing; reconfiguration

1 引言

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,大数据的处理模式也正从批处理向流处理发生转变[2].值得注意的是,分布式的流处理系统和集中式的流处理系统都会碰到单点瓶颈问题,针对这个问题,不少工作已经提出了相应的解决方案[3,4],总体来看,它们可以被分为两类,一类是以减少时间复杂度为宗旨的解决方案,另一类是以减少空间复杂度为宗旨的解决方案.前者的典型代表是卸载技术[5],当节点的处理能力无法满足当前的处理负载时,卸载技术会通过丢弃部分待处理的数据来降低这个节点的处理负载,至于哪部分数据被丢弃则取决于该数据对查询结果的影响度.后者的典型代表包括概要技术,直方图技术和小波变换技术[6],这些技术通过多个元组的聚合查询的近似结果来减少存储开销.然而,上述两类技术都会丢失部分数据,因此,人们开始试图寻找一种避免负载突然变化导致数据丢失的解决方案,这可以看成流处理系统(Stream Processing System,SPS)从分布式演化到并行分布式的一个重要动机.在并行分布式

收稿日期:2013-09-30;修回日期:2014-03-30;责任编辑:孙瑶
基金项目:国家863高技术研究发展计划(No.2011AA010703);中国科学院战略性先导专项(No.XDA06030602);国家自然科学基金(No.61402464);中国博士后基金(No.2013M541076)

的流处理系统中[7],一个查询的任意一个算子都可以部署到多个节点上并行执行,有效避免了单点瓶颈.然而,在很多情况下,数据流中的数据往往来自分布在不同地理位置的数据源,因此并行分布式 SPS 会面临以下两个挑战.

第一个挑战是并行化处理技术.这种并行化技术必须要保证语义透明性,也即是由一个并行分布式查询所产生的结果要等同于由集中式或者分布式查询所产生的结果.在这方面,伯克利大学的 Tyson Condie 等人[8],威斯康星大学的 Wang Lam 等人[9],加州大学的 MateiZaharia 等人[10]分别提出了并行处理模型,同时,工业界也提出了 Storm[11]和 S4[12],然而,这些并行处理模型没有区分有状态算子和无状态算子,导致这些算子在并行处理过程中会产生很多通信开销.为此下面以图 1(a)所示的查询为例进行比较,这个查询由 2 个有状态的算子(连接 J 和聚合 Ag)以及 4 个无状态的算子(两个映射 M 和两个过滤 F)组成.这个查询部署在 90 个节点组成的集群上.其中无状态算子不会保存元组的状态,并且节点可以独立的处理每个元组.有状态算子需要保存元组的状态,执行计算时需要其他元组的参与,因此,笛卡尔积算子 CP 也是有状态算子.在图 1(b)中,并行化单元是单个算子,每个算子部署在一个子集群上,每个子集群有 15 个节点,由一个子集群到下一个子集群的所有节点都会进行通信.总的跳数等于算子数减 1,也就是 5,每个节点的扇出则是 15,总的扇出数是 15^5.在图 1(c)中,算子集并行不仅减少了跳数也减少了扇出数.其原理是,为了保证语义透明,通信只发生在有状态运算之前,所以并行化单元是有状态算子,因此每个查询被划分的子查询个数等于有状态算子个数加 1.在图 1(c)中,该查询有两个有状态算子和一个无状态算子,所以它被划分成 3 个子查询,每个子查询被分配到一个由 30 个节点组成的集群上.第一个子查询包括无状态算子,而后面的两个子查询都是由一个有状态算子和其后的无状态算子组成.总的跳数等于有状态算子的个数,也就是 2,如果查询没有无状态算子则减 1.同时,每个节点的扇出则是 30,总的扇出数是 30^3.

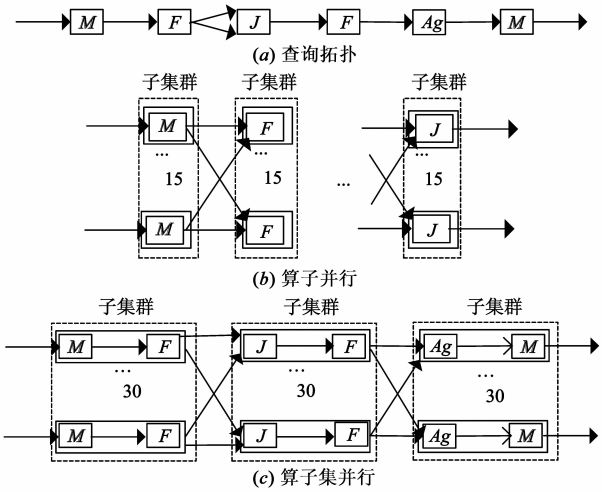


图1 并行化技术比较

第二个挑战是可扩展技术.现有的大部分 SPS 都是静态部署的,也就是一旦查询被部署到 SPS 中就无法改变.由于数据流本身具有高度可变的性质,这样的静态部署方式是不合适的.为了克服这种缺陷,并行分布式的 SPS 需要具有可扩展性,也就是说并行分布式的 SPS 需要根据当前负载动态增加或者减少节点以实现在保证服务质量的前提下尽可能地减少计算资源.为此文献[13]中提出了一种可扩展协议,该协议根据节点负载动态增加或者减少流处理线程.在这个协议中,一个队列用来维护节点的输入元组,同时多个流处理线程并行处理队列中的元组.然而,这个协议有两点不足:第一点是当出现大量并发的线程时,多线程系统无法很好地进行扩展.第二点是整体的处理性能会受单个物理机器资源的限制.为此文献[14]又提出了一些 SPS 可扩展的设计方案.第一个设计方案是当定义一个查询的划分时,并行编译器会给出一个最大化吞吐量的子集群,该子集群能够降低垂直负载不均;第二个设计方案是通过负载均衡来解决水平负载不均,其中负载均衡会把一个正在执行查询的节点负载重新分配给另一个负载较少的节点.如果负载无法在已有节点中实现均衡,那么就会增加新的节点,正在执行的查询也会重新部署.上面这个工作表明,可扩展要与负载均衡相结合,这意味着,只有当已经分配的节点无法通过负载均衡来满足当前数据流对处理性能的要求时才会增加新的节点.为此本文提出了一个支持可扩展的并行分布式流数据处理平台—流水行云,其中的主要贡献如下:

(1)提出了一种在 shared-nothing 架构下的算子集并行化处理技术.该技术区分了有状态算子和无状态算子,并且通过有状态算子的分发器和收集器实现了数据流的保序,保证了语义透明性,同时最大程度上减少了算子并行处理过程中的通信开销.

(2)提出了结合负载均衡和重配置的可扩展技术.当运行节点的处理负载分配不均衡时,负载均衡将会被触发.当已经分配的所有节点都无法处理输入负载时,流水行云会自动增加新的节点,重配置将会被触发.

2 体系架构

流水行云的整体架构如图 2 所示,其中一个查询定义为一个有向无环图,并且图中每个节点代表一个 Spring 或者 Processor 组件,其中,Spring 负责产生流数

据,也即是元组,Processor 负责处理流数据.图中的边表示数据流向.Processor 定义了类似于关系代数操作的查询算子,它们被分为有状态算子和无状态算子,下面介绍图 2 右边所示的查询处理组件.

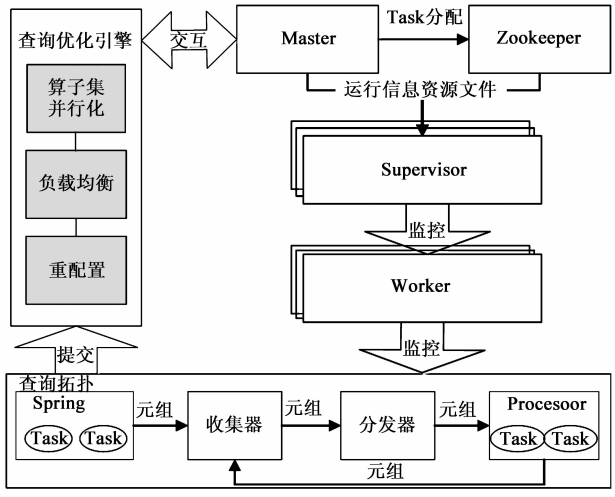


图2　流水行云的整体架构

Task:Task 是执行 Spring 或者 Processor 的线程,主要任务是将输入数据流中的元组进行转换并输出.

Worker:Worker 是一个 Java 虚拟机进程,可以看作监控正在运行的一个或者多个 Task 的容器.

Supervisor: Supervisor 是一个监控进程,负责监控 Worker 的状态,这里一个 Supervisor 仅负责监控一个 Worker.

Master:Master 是一个管理进程.当描述查询的 XML 文件和执行查询所需的代码提交到集群之后,Master 开始进行任务分配.

查询优化引擎:根据有状态算子将查询拓扑划分为并行处理的子查询,同时,通过重配置协议和负载均衡协议,使得 Master 能够根据输入负载动态调整处理节点的负载和个数.这两个技术是流水行云区别于其他流处理系统的主要特点.

3　并行化技术

流水行云根据有状态算子个数将提交的查询划分成多个子查询,子查询中包含有状态算子则称为有状态子查询,否则称为无状态子查询,并且执行有状态子查询的 Task 都会有一个分发器 ST 和收集器 RT,它们用来实现数据流的保序,而执行无状态子查询的 Task 仅有一个分发器.当查询提交到集群中执行时,可以将集群逻辑划分为若干个子集群,并将不同的子查询分配到这些逻辑划分的子集群中并行执行.为了保证子集群中的所有元组都能够被正确地发送到后继子集群中,前驱子集群中各个 Task 处理完的元组需要根据分配策略被分配到不同的元组桶中.这里的分配策略是指对元组的属性值或是属性组合的值取模,将值相同的元组分配到相同的元组桶中.假设元组 t 和 b 个元组桶,通过元组的一个或多个属性值将元组 t 分配到某一元组桶中,这样当所有的元组被分配到不同的元组桶之后,同一个元组桶中的元组将会由后继子集群的同一 Task 处理.为了将前驱子集群产生 b 个元组桶的元组都分配到后继子集群中 n 个 Task 中处理,每个子集群设置了桶映射代理 BMP,这个代理的作用就是将前驱子集群中的某个元组桶与后继子集群中的某个 Task 相关联.BMP[b].dest 的值可以唯一确定后继子集群中的某一个 Task,并且该 Task 接收并处理所有 b 元组桶中的元组.例如 BMP[b].dest = Task A,可以理解为后继子集群中的 Task A 接收并处理 b 元组桶中的所有元组.

3.1　分发器

分发器是将本地子查询所产生的元组分发给后继子集群中的各个 Task 中.其中对于有状态算子,桶编号相同的元组必须由同一个 Task 接收并处理,那么,前驱子集群中有状态子查询的分发器必须具有语义感知的功能,特别是它必须了解后继子集群的状态操作语义.下面重点考虑笛卡尔积的分发器.

笛卡尔积是将两个集合中任意某个元组与另外一个集合中的所有元组形成一对组合,而没有一个明确的连接条件,所以该算子在分布式系统中实现是比较复杂的.笛卡尔积算子的实现如下:前驱子集群中的两个分发器 ST_0和 ST_1分别将元组映射到 b_0和 b_1桶中,分发器中的 BMP 会为$(b_0, b_1) \in B_0 \times B_1$选择后继子集群中的接收者.具体实现方式如下:假设根据元组属性 A_i和A_j将元组映射到不同的元组桶中.当元组进入到前驱子集群的 ST_0时,根据 $b_0 = H(A_i \cdots A_j) \bmod B_0$可以计算出元组将会进入的元组桶序号.同样,当元组进入到前驱子集群的 ST_1时,根据 $b_1 = H(A_i \cdots A_j) \bmod B_1$可以计算出元组将会进入的元组桶序号.最后,根据 BMP[b_0, b_1].dest 将元组桶中的元组发送给后继子集群中的 Task 中.图 3(a)描述了一个包含两个 M 和一个 CP 的查询.图中的最上方 ts 为时间戳,M_l对输入元组进行除法运算并将结果输出.M_r是将小写字母转换成大写字母并将结果输出.CP 的条件是 left.number mod 2 = 1 && right.letter! = A and interval≤2,其中只有 3 和 1 符合 left.number mod 2 = 1 的条件,同时 right.letter! = A 的条件将会把 A 排除,仅保留 B、C 和 D.时间窗口 interval≤2 的条件会得到最终的查询结果是(3, D),(3, C),(1, C),(1, B).图 3(b)描述了该查询被划分为 1 个无状态子查询和 1 个有状态子查询的并行化处理过程.其中,

无状态子查询被部署到包含 2 个节点的子集群 1 中,每个节点包含 2 个映射操作 M_l 和 M_r,每个映射操作都会包含 1 个分发器用来将元组映射到不同的元组桶中,并将元组桶中的元组发送到后继子集群中的接收 Task.有状态子查询被部署到包含 4 个节点的子集群 2 中,每个节点有 1 个 Task 执行 1 个 CP,在每个 CP 之间都会有 2 个收集器,例如 RT_{l_0} 和 RT_{r_0},每个收集器都接收某一元组桶中的元组.为了实现流数据的并行处理,初始的数据流需要被划分为多份.同时,当初始的数据流被划分为 n 份,那么后继子集群中接收的 Task 的个数必须等于 n^2.如图 3(b)所示,每个初始的数据流被划分为 2 份,分别由子集群 1 中的 2 个节点处理,后继子集群 2 中需要 4 个 Task 来同时处理子集群 1 中输出数据.从图 3(b)中可以看出元组 0 和元组 3 被发送到节点 0 和节点 2 中的某个 Task 处理,元组 1 和元组 2 被发送到节点 4 和节点 6 中的某个 Task 处理,元组 A 和元组 C 被发送到节点 1 和节点 5 中的某个 Task 处理,元组 B 和元组 D 被发送到节点 3 和节点 7 中的某个 Task 处理.因此,子集群 2 中的每个节点都处理初始的数据流的四分之一,有效地提高了处理性能和吞吐量.

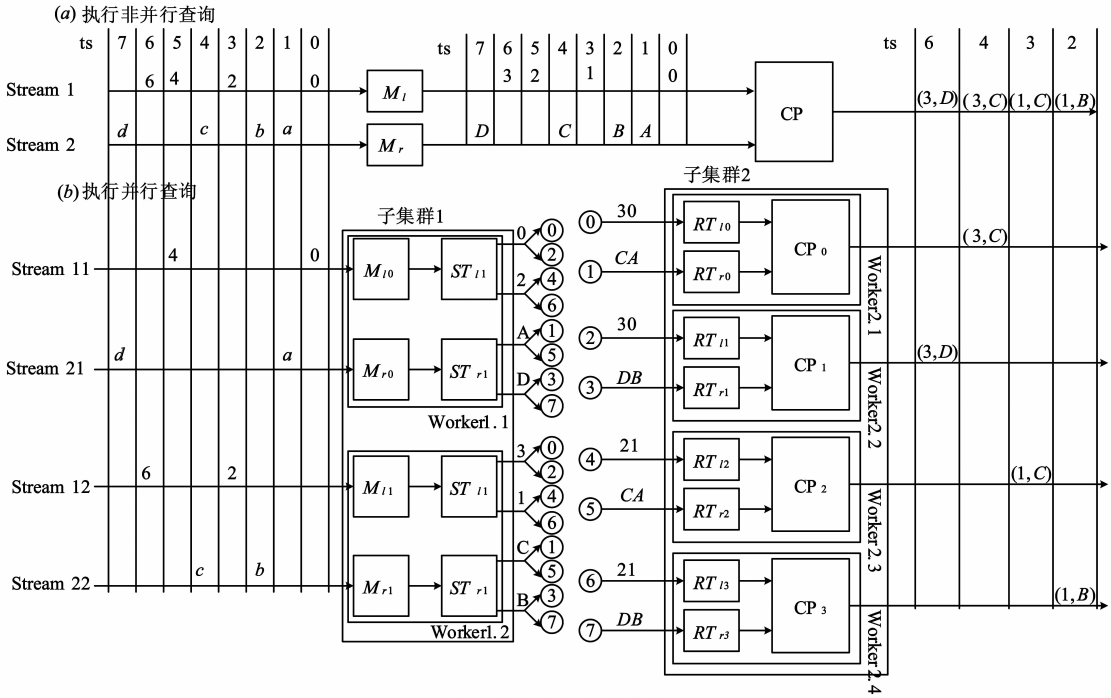


图3 笛卡尔积的分发器

3.2 收集器

与分发器一样,收集器也需要保证语义透明性.一个简单收集器仅将前驱子集群中分发器发送的元组发送到对应的 Task,这样可能会导致元组处理产生不正确的结果.

如图 4(a)所示的一个按照时间顺序接收输入元组(P, Q, X, Y)和(2,3,5,7)的 CP 查询.非并行的 CP 查询使用了滑动窗口,滑动窗口的最大值是预先定义的,并且会在执行过程中动态调整.如图 4(b)所示,时间戳小的在窗口的最右边.当上(下)边的输入流收到元组 t 时,下(上)边的窗口会立刻更新,将一些 t'-t 大于等于窗口大小的元组 t 移除.如图 4(b)中的步骤 6 所示,当元组 5 到达时,P 立刻从窗口中被删除,在步骤 7

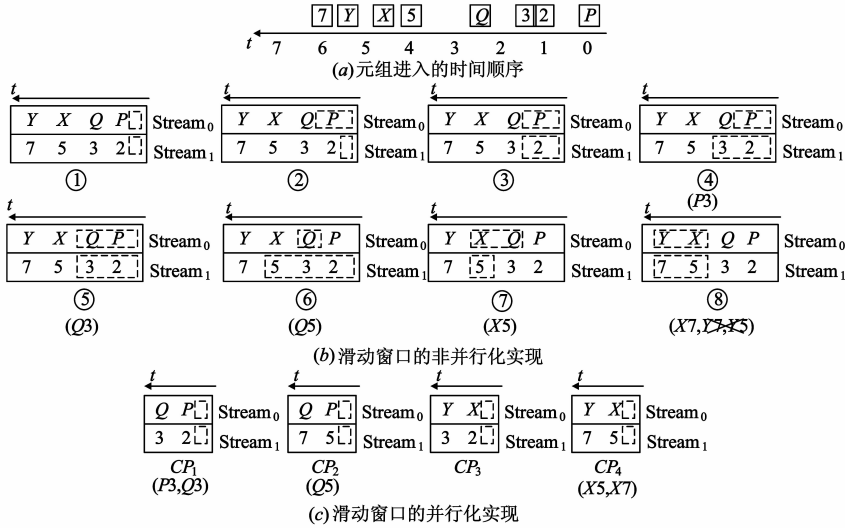


图4 滑动窗口的收集器

时,元组 X 到达后元组 2 和元组 3 立刻被删除.然而,由于并行化处理需要将初始的数据流划分多份,因此元组会以随机顺序分配给收集器,导致笛卡尔积的滑动窗口处理结果出错.如图 4(c)中 CP_4,假如 CP_4收到元组的顺序是 X、Y、7、5,元组 7 的进入会使得元组 X 从窗口中删除,那么在元组 B 进入到窗口后将不会与元组 X 相匹配产生 $X5$ 结果,因此如果以上述的顺序接收元组将会导致输出结果错误.要想保证输出结果正确,则需要输入元组以正确的时间顺序进入到滑动窗口,收集器的作用就是将输入元组按照时间顺序排序,然后再将接收的元组交给子查询来继续处理,那么子查询将会以时间顺序产生正确输出结果.为了避免收集器被阻塞,如果前驱子集群中的每个分发器在过去的 d 时间内一直是空闲的,并且没有处理过任何元组,那么它将会发送一个哑元元组给后继子集群中的收集器,收集器在处理元组的时候会将哑元元组丢弃,这样收集器将会一直处理元组,不会因为没有元组被处理而导致阻塞.滑动窗口的改进方式如图 4(c)所示,每个节点的子查询都并行处理输入元组,虽然各个分发器对输出元组没有实现时间上的同步,但是收集器实现了处理元组在时间上的有序,保证了并行化技术的语义透明性.

4　可扩展技术

支持可扩展的目的是使得 CPU 平均利用率尽可能接近目标利用率(TUT),从而达到最佳的流处理性能,为此本文提出了重配置协议和负载均衡协议.当 CPU 平均利用率超过利用率阈值上界(UUT)或者低于利用率阈值下界(LUT),需要增加或者减少节点时,重配置协议会被触发.当 CPU 利用率的标准差大于最大非平衡阈值(UIT)时,负载均衡协议会被触发.

4.1　重配置协议

在重配置协议中,Supervisor 进程会向前驱子集群中的所有分发器发送重配置命令 ReconfigCommand(Task1,Task2,元组桶),该命令用于改变某些元组桶中元组的去向,将原本要发送到 Task1 的元组桶中的元组,发送到 Task2 中.同时,为了确定重配置时间 startTS,子集群中参与重配置 Task 的时间戳会被设置.当前驱子集群中的分发器收到重配置命令时,它会向需要重配置的 Task 发送控制元组,控制元组携带的信息为各个分发器收到重配置命令后所发送的最后一个元组的时间戳,参与重配置 Task 会从多个控制元组中选取最大的时间戳作为 startTS.分发器主要维护 BMP[b]的三个信息,其中 BMP[b].endTS 表示重配置的结束时间,BMP[b].newTask 表示将要转发的新的 Task,BMP[b].status 用来区分正常处理元组还是执行重配置命令.

4.1.1　窗口重构协议

窗口重构协议目的是减少重配置过程中元组在不同 Task 之间切换的通信开销,并且只有当原来的 Task 处理完窗口中的元组后,元组才会只发送给新的 Task,而不会再发送给原来的 Task.从协议 1 的 6 – 17 行可以看出 Task A 会处理出现在 BT[b].startTS 之前的所有元组,而 Task B 会处理 BT[b].startTS 之后的所有元组.在 startTS 之后的某一时间段内的元组需要同时发送到 Task A 和 Task B,而 Task A 通过开始时间、窗口大小和增量大小计算出 Task A 处理元组的结束时间 BT[b].endTS,同样 Task B 会以同样方式计算出 BT[b].switchTS,Task A 会将BT[b].endTS通过 EndOfReconfiguring 发送给前驱子集群中的分发器,在 BT[b].endTS之后的所有元组只会发送给 Task B.

协议 1　窗口重构协议

```
分发器
前提条件:将收到的元组 t 发送到元组桶 B 中
1:if BMP[b].status = reconfiguring && t. ts > BMP[b].endTS then
2:BMP[b].dest = BMP[b].newOwner
3:BMP[b].status = normal
4:将元组 t 发送到 BMP[b].destTask
前提条件:收到 EndOfReconfiguration(b,ts)
5:BMP[b].endTS = ts
Task A
前提条件:BT[b].ownership = old
6:BT[b].endTS = ComputeEndTS()
7:向前驱子集群中的分发器发送 EndOfReconfiguration(b, BMP[b].endTS)
前提条件:收到元组桶 B 的元组 && BT[b].ownership = old
8:if t. ts < BT[b].endTS then
9:处理元组 t
10:else
11:  丢弃元组 t
12:  BT[b].ownership = notOwned
TaskB
前提条件:BT[b].ownership = new
13:BT[b].switchTS = computeSwitchTS()
前提条件:接收到元组桶 B 的元组 t && BT[b].ownership = new
14:if t.ts < BT[b].switchTS then
15:  丢弃元组 t
16:else
17:  BT[b].ownership = owned
{从此开始处理元组桶 B 中的所有元组}
```

4.1.2　状态重构协议

窗口重构协议能够减少元组在 Task 之间切换的通信开销,但是该协议的一个不足就是协议的完成时间依赖于窗口大小,因此如果窗口很大的话,那么窗口重构协议的执行时间会很长,这是 SPS 所不能接受的.为

此状态重构协议被提出.该协议不受窗口大小的影响,它通过一个检查点来记录元组桶,该检查点就是元组发送到新的 Task 和原来的 Task 的分界点.同样,通过向 Task A 和 Task B 发送控制元组来确定 BT[b].startTS,并且在发送控制元组之后,前驱子集群中的所有分发器发送的元组都要同时发给 Task A 和 Task B.Task A 就向前驱子集群中的所有分发器发送 EndOfReconfiguration 命令(并将 startTS 赋值给 endTS),此时 Task A 会将时间大于等于 startTS 的元组丢弃,并将检查点的状态在序列化后发送给 Task B.前驱子集群中的分发器在收到 EndOfReconfiguration 后,会将时间戳小于 endTS 的元组发送给 Task A 和 Task B,大于等于 endTS 的元组仅发送给 Task B.Task B 在没有收到 Task A 发送的检查点之前会将接收到的所有元组缓存起来,当接收到 Task A 发送的序列化的检查点后,Task B 开始处理所有缓存的元组,完成重配置过程.状态重构协议的执行过程与窗口重构协议相似,不同的是 BT[b].startTS 和 BT[b].endTS 是在同一个时间点,并且把出现在 endTS 之前的一部分元组同时发给 Task A 和 Task B,而把出现在 endTS 之后的元组只发给 Task B.

4.2 负载均衡协议

随着负载不断增加,一次增加一个节点很可能无法使得 CPU 的平均利用率降低到目标利用率之下,所以还会继续增加节点.为了避免这种级联增加节点现象,本文提出的负载均衡协议会根据集群大小和当前负载来计算还需要增加多少节点才能使得 CPU 的平均利用率接近目标利用率.

在负载均衡协议中,首先,Master 进程周期性地收集子集群中 Worker 的相关信息,该信息包括每个 Task 的 CPU 平均利用率 U_i、某个元组桶每秒钟处理元组的数量 T_b.然后,Master 进程根据以上信息计算每个子集群 CPU 的平均利用率 U_{av},假如 U_{av} 的值不在利用率阈值的上界与下界之间,Master 进程会根据当前的 Task 数量和输入负载计算实际需要 Task 的数量.当计算所得的 Task 数量大于当前 Task 数量,则需要向集群中分配新 Task,否则,减少 Task 数量.在减少 Task 之前应该通过卸载技术将要减少的 Task 的输入负载移植到其他 Task 来处理.当集群 CPU 利用率的标准差 U_{sd} 大于最大非平衡阈值 UIT 时,开始执行 BalanceLoad.BalanceLoad 是基于贪心策略的负载均衡算法,它首先根据 CPU 利用率对 Task 进行排序,然后对排序后的 Task,根据其中的元组桶每秒处理元组的数量 T_b 再对元组桶进行排序,这样每次迭代都会产生最大(最小)输入负载的 Task 和处理元组速度最快(最慢)的元组桶 T_b,最后将输入负载最大的 Task 中的 T_b 值最大的元组桶分配给输入负载小的 Task 上,直到 CPU 利用率的标准差 U_{sd} 的提高值小于最小改善阈值,也即使重配置所要达到的最低性能提高值,其中迭代终止.上述分析表明排序是该算法的主要步骤,因此时间复杂度是 $o(n\log(n)+m\log(m))$,其中 n 和 m 表示 Task 和元组桶 B 的个数.

协议 2 负载均衡协议

```
1:计算 CPU 平均利用率 U_av
2:if U_av > UUT && U_av < LUT then
3:old = n
4:n = ComputeNewConfiguration(TUT,old,U_av)
5:  if n > old then
6:    Provision(n-old)
7:  if n < old then
8:    freeNodes = Offload(old-n)
9:    Decommission(freeNodes)
10:计算 CPU 利用率标准差 U_sd
11:if U_sd > UIT then
12:    BalanceLoad(U_sd)
```

5 实验

本实验在由 4 台服务器启动 60 个 Worker 进程模拟的 60 个节点组成的 shared-nothing 架构的集群上进行的.每台服务器配置 16 个内核,16GB 内存,10 千兆位以太网以及 1TB 的硬盘空间.所有的实验都经过三个阶段:预热、稳定和冷却.实验数据是在持续至少 10 分钟的稳定阶段后测量得到的.每个实验都运行三次并且取其中的平均值作为报告结果.实验使用图 5 所示的查询,其中的输入元祖是以时间变化的正态分布来产生的 10000 个不同手机号码,其中平均值 μ,标准差 σ.

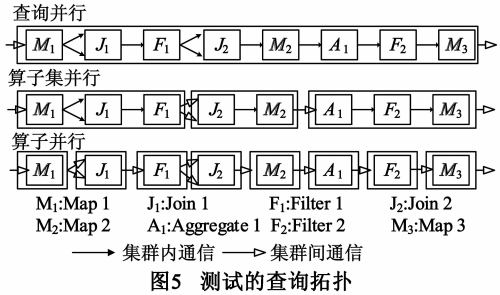


图5 测试的查询拓扑

5.1 扩展性

本实验主要验证输入负载对可扩展性的影响.首先,我们验证并行化策略的可扩展性,其中查询的并行实例数是 30.

图 6 显示了不同并行策略下的 CPU 在元组处理开销、通信开销以及空闲之间的分配比例.其中,并行策略需要集群中每个执行的 Task 都要和其他所有执行的 Task 进行通信,其中每一个有状态算子,它们的通信次

数大于 3×30^2.图 6 显示了查询并行策略的通信开销在 40%左右,剩下 60%被用来进行元组处理.算子并行策略显示的通信开销大于 30%,同时元组处理开销大约在 35%.CPU 中的空闲时间与理论计算的 Task 个数和实际 Task 个数的差值有关.例如,当计算一个最优分配方案时,一个算子或许需要 4.3 个实例并行执行,但是实际的实例个数为 5,因此导致一个 CPU 没有充分被利用.算子集并行策略显示了最低的通信开销,大约 10%,如同前面的策略一样,理论计算和实际需要的 Task 数量的差异会导致资源未被充分利用.然而,由于资源的使用数量较少,因此空闲时间会低于算子并行策略.

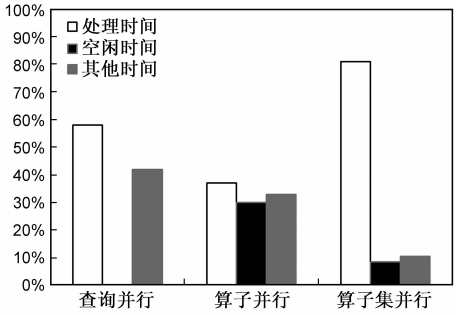


图6　算子集并行的CPU利用率测试

图 7 的上半部分显示了并行实例增多时的三种并行策略的可扩展性,对于查询并行策略,我们只使用一半 Task,这是因为 Task 的数量超过 30 个时,它的扇出开销就已经超过了部署时的可用资源.对于每种策略,使用不同的集群环境来测试,然而,我们只记录达到最高吞吐量的集群配置.算子集并行分别达到了算子并行和查询并行的性能的 2.5 到 5 倍.图 7 的下半部分显示了输入负载增加时的 CPU 利用率.查询并行和算子集并行方法达到了 100%的 CPU 利用率.然而,前者在较低的输入负载时达到了最大 CPU 利用率(≤10,000 元组/秒),而算子集并行方法此时仅达到 60%的 CPU 利用率,大约 40%的 CPU 没有被使用.因此算子集并行能够更好适应输入负载增多的情况,主要原因是重配置协议和负载均衡协议保证了 CPU 平均利用率尽可能接近目标利用率.

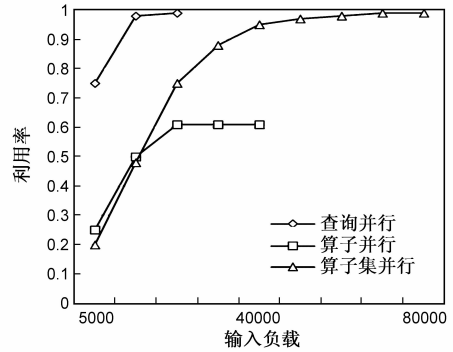
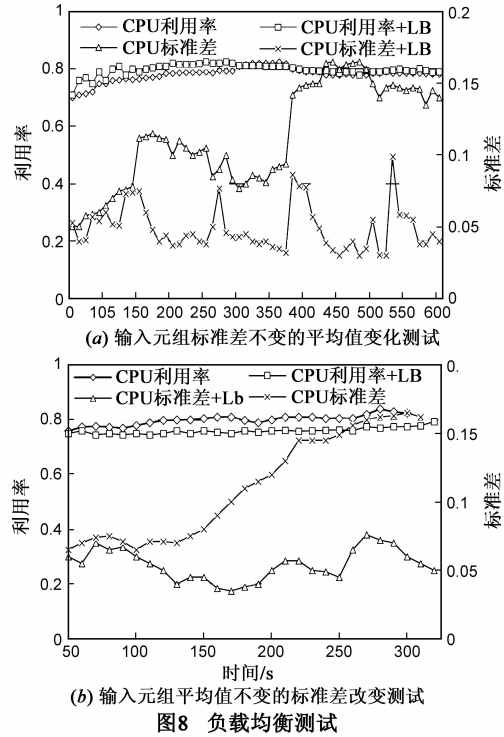


图7　算子集并行的可扩展测试

5.2　负载均衡

本实验的目标是验证输入负载变化时的负载均衡效果.我们仍然使用图 5 中的查询,并且监控由输入负载 150000t/s 的 15 个节点组成的集群.根据时间变化的正态分布产生输入元组.目标是当输入负载变化时,启动负载均衡可以使得流水行云在已经分配的节点中保持均衡的 CPU 利用率.图 8(a)比较了启动负载均衡和没有启动负载均衡的性能.这个实验分为几个阶段,在第一个阶段,输入负载满足均匀分布.在其他阶段,输入负载满足 μ 周期性从 2000 改变到 8000 的正态分布,在图 8(a)中,对于每个阶段,μ 和 σ 被指定,菱形显示了负载均衡没有启动时,CPU 平均利用率和它的标准差.在这个情况下,每次 μ 改变时,σ 都在增长.正方形显示了启动负载均衡时的性能.在每个新阶段启动初期,σ 都呈现出峰值,然后在启动负载均衡后开始下降.负载均衡保证当输入负载变化时,σ 一直稳定.由于输入负载固定,两种配置都显示了稳定的 CPU 平均利用率.图 8(b)显示了 μ 保持不变时,σ 在 25 步内周期性地从 100 降低到 25 时的测试,如果没有负载均衡,随着 σ 的降低,并行实例之间的不均衡程度会加大.如果有负载均衡,负载被重新分配,并且 σ 一直保持在最大不均衡阈值之下.因此负载均衡可以有效地根据输入负载调整处理节点,提高资源利用率.

实验表明,SPSPS 不仅提供了保持查询语法和语义



(a) 输入元组标准差不变的平均值变化测试

(b) 输入元组平均值不变的标准差改变测试

图8　负载均衡测试

透明的并行化技术,而且通过减少并行执行查询的Task之间的通信开销提高了其可扩展能力,同时通过重配置协议实现的负载均衡减少了不同输入负载条件下处理的资源开销.

6 结束语

本文提出了一个支持可扩展的并行分布式流处理系统—流水行云,该系统的并行化技术不仅提供了和集中式处理相同的结果,而且通过查询拓扑的有效划分最大程度减少并行处理的通信开销.同时,结合负载均衡和重配置的可扩展技术使得该系统能够根据输入负载动态调整处理节点的负载和个数,提高资源利用率.实验表明了该系统扩展能力的有效性.未来我们希望该系统能够进一步支持复杂的查询算子,并且提供多个数据流的共同查询.

参考文献

[1] 孟小峰,慈祥.大数据管理:概念,技术与挑战[J].计算机研究与发展,2013,50(1):146 – 169.
Meng Xiao-feng, Ci Xiang. Big data management: concepts, techniques and challenges[J]. Journal of Computer Research and Development, 2013, 50(1): 146 – 169. (in Chinese)

[2] R Kumar. Two Computational Paradigm for Big Data[OL]. http://kdd2012.sigkdd.org/sites/images/summerschool/Ravi.Kumar.pdf, KDD summer school, 2012.

[3] 孙圣力,戴东波,黄震华,张齐勋,周立新.概率数据流上Skyline查询处理算法[J].电子学报,2009,37(2):285 – 293.
Sun Sheng-li, Dai Dong-bo, Huang Zhen-hua, Zhang Qi-xun, Zhou Li-xin. Algorithmon computing skyline over data stream[J]. Acta Electronica Sinica, 2009, 37(2): 285 – 293. (in Chinese)

[4] 钱江波,王永利,陈征,陈华辉,金光.数据流窗口连接查询处理器研究[J].电子学报,2009,37(2):404 – 409.
Qian Jiang-bo, Wang Yong-li, Chen Zheng, et al. Hardware processor for window joins over multiple data streams[J]. Acta Electronica Sinica, 2009, 37(2): 404 – 409. (in Chinese)

[5] B Brain, D Mayur, M Rajeev. Load shedding for aggregation queries over data streams[A]. Proceedings of the 20th International Conference on Data Engineering[C]. USA: IEEE, 2004. 350 – 361.

[6] D Mayur, M Rajeev. The sliding-window computation model and results[A]. Data Streams[C]. USA: Springer US, 2007, 31: 149 – 167.

[7] K H Lee, Y J Lee, H Choi, Y D Chung, B Moon. Parallel data processing with MapReduce: a survey[J]. ACM SIGMOD Record, 2012, 40(4): 11 – 20.

[8] C Tyson, C Neil, A Peter, et al. MapReduce online[A]. Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation[C]. USA: ACM, 2010. 21 – 23.

[9] L Wang, L Lu, P STS, et al. Muppet: MapReduce style processing of fast data[J]. Proceedings of the VLDB Endowment, 2012. 5(12): 1814 – 1825.

[10] M Zaharia, T Das, H Y Li, et al. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters[A]. Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing[C]. USA: ACM, 2012. 10 – 10.

[11] Twitter. Storm Project[OL]. http://storm-project.net/, 2011-08-16.

[12] L Neumeyer, B Robbins, A Nair, A Kesari. S4: Distributed stream computing platform[A]. Proceedings of IEEE International Conference on Data Mining Workshops (ICDMW)[C]. USA: IEEE, 2010. 170 – 177.

[13] S Scott, A Henrique, G Bugra, et al. Elastic scaling of data parallel operators in stream processing[A]. Proceedings of IEEE International Symposium on Parallel & Distributed Processing[C]. USA: IEEE, 2009. 1 – 12.

[14] T Heinze. Elastic complex event processing[A]. Proceedings of the 8th Middleware Doctoral Symposium[C]. USA: ACM, 2011. 1 – 6.

作者简介

张 鹏 男,1984 年出生,安徽淮南人.2013 年在中国科学院计算技术研究所获工学博士学位.现为中国科学院信息工程研究所博士后,主要从事数据流处理及云计算方面的研究.
E-mail: pengzhang@iie.ac.cn