面向大数据的分布式流处理技术综述

张 鹏, 李鹏霄, 任 彦*, 林海伦, 杨 崚, 郑 超

(中国科学院 信息工程研究所 北京 100091)
(信息内容安全技术国家工程实验室 北京 100091)
(国家计算机网络应急技术处理协调中心 北京 100029)
(中国科学院大学 北京 100049)
(pengzhang@iie.ac.cn)

Distributed Stream Processing and Technologies for Big Data: A Review

Zhang Peng, Li Pengxiao, Ren Yan*, Lin Hailun, Yang Rong, Zheng Chao
(I*nstitute of* I*nformation* E*ngineering* CAS, B*eijing* 100091)
(N*ational* E*ngineering* L*aboratory for* I*nformation* S*ecurity* T*echnologies*, B*eijing* 100091)
(N*ational* C*omputer* N*etwork* E*mergency* R*esponse and* C*oordination* C*enter*, B*eijing* 100029)
(U*niversity of* C*hinese* A*cademy of* S*ciences*, *Beijing* 100049)

Abstract The era of Big Data has begun, the users are more eager for fresh and low-latency processing results than ever. For this reason, this paper reviews the recent stream processing models for Big Data and focuses on the parallel-distributed processing models, and presents their design goals and architectures. Moreover, this paper discusses the main challenges in designing a parallel-distributed stream processing model and future works.

Key words data stream; parallel-distributed; load balancing; fault-tolerance; big data

摘 要 随着大数据的到来，数据流处理技术又成为了新的研究热点。为此本文回顾了近期提出的面向大数据的流处理技术的现状，并且从流处理模型上对这些技术进行了划分，重点分析了面向大数据的并行分布式的流处理模型的设计目标和架构。同时，重点讨论了并行分布式流处理模型的关键技术以及未来技术的展望。

关键词 数据流；并行化；负载均衡；故障容错；大数据

中图法分类号

1 引言

数据流处理技术一直是数据库领域的一个研究点。早期的数据流处理系统(Stream Processing System, SPS)是一种集中式的架构。典型的代表包括Aurora[1][2]，STREAM[3][4][5]，TelegraphCQ[6][7]。在这些SPS中，一个查询全部部署在一个节点上。这种集中式的系统一个最大缺点是一旦这个节点的资源出现饱和(例如CPU资源或者内存资源)，那么查询处理的速度就会减慢，最终导致查询结果的输出延迟变长。因此随后出现了从集中式的SPS向分布式的SPS的演化。第一个分布式的SPS的典型代表是Borealis[8][9]，它是Aurora和Medusa的改进版本[10][11]。

文献[12]把查询分为两种并行执行方式：查询间并行(inter-query)和查询内并行(intra-query)，其中前者是指多个查询在不同的节点上并行执行，后者是指一个查询在多个节点上并行执行。查询内并行可以进一步分为算子间并行(inter-operator)和算子内并行(intra-operator)，其中inter-operator是指属于同一个查询的不同算子在不同节点上并行执行。intra-operator是指一个查询的每个算子在多个节点上并行执行。除了支持在一个节点上执行一个完整的查询，集中式的SPS也支持查询间并行，但是不支持查询内并行，而分布式的SPS则支持查询内并行，但是无法提供算子

*通讯作者

基金项目:本课题得到国家自然科学基金(No.61402464),中国博士后基金(No.2013M541076),国家242信息安全专项(2013F107)资助

内并行，而并行分布式 SPS 则提供了算子内并行。

值得注意的是，分布式的 SPS 可能和集中式的 SPS 碰到相同的问题：一旦某个节点的负载突然出现高峰，整个查询结果的输出延迟就会变得很长，我们把这种情况称为单点瓶颈，其中的主要原因在于每个数据流只经过一个节点进行处理，当数据流的规模超过这个节点处理能力的上限时，节点的处理时间便增多，导致整个查询结果的输出延迟增长。针对这个问题，不少工作已经提出了相应的解决方案，总体来看，它们可以被分为两类，一类是从减少时间复杂度出发的解决方案，另一类是从减少空间复杂度出发的解决方案。前者的典型代表是卸载技术[13][14]，当节点的处理能力无法满足当前的处理负载的要求时，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[15][16]。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。例如，当服务质量标准(Quality of Service, QoS)被定义后，卸载技术会在尽可能保证服务质量的情况下丢弃对服务质量影响小的数据。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[17]，这些技术旨在减少节点的存储开销，例如文献[18]提出的指数直方图通过模拟多个元组上的聚合查询的近似结果减少了需要存储的数据信息。

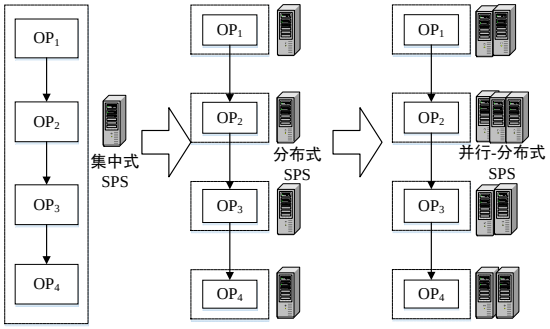

图 1 数据流处理模型的演化

然而上述两类技术在数据流处理的过程中都会丢失部分信息，因此人们开始试图寻找一种避免负载突然变化导致丢失信息的解决方案。我们将这个动机看成是 SPS 从分布式演化到并行分布式的一个主要因素。在并行分布式的 SPS 中，一个查询的任意一个算子都可以部署到多个节点上并行处理。这种设计思想的主要目的是为了避免一个数据流都集中到一个节点上进行处理，从而导致这个节点出现单点瓶颈。图 1 给出了 SPS 的演化过程，其中描述了一个由 4 个算子组成的查询在集中式的 SPS、分布式的 SPS 和并行分布式的 SPS 的处理过程。

随着大数据时代的到来[19]，对已有数据的利用率的高低直接影响企业的经济效益，因此业界也逐渐开始意识到高效的数据查询在帮助企业进行市场决策，提高企业生产力的重要性，为此，大数据的处理模式也从批处理向流处理发生转变[20][21]，并行分布式的 SPS 又成为了研究热点。下面，我们回顾一下其中的一些代表性的工作。

2 并行分布式处理系统

Storm[22]是一个并行分布式的数据流处理系统。Storm 的设计重点放在了分布式、并行和故障容错上，而如何处理元组则由用户定义，这可以看成是数据流上的 MapReduce[23]。在 MapReduce 模型中，用户负责定义读取和产生数据以及并行处理这些数据的函数，而系统负责管理执行这些函数的多线程的实例。Storm 的查询也是使用箭框图来表示，其中包含两个对象：Spout 和 Bolt。Spout 负责产生数据流，Bolt 负责处理输入的数据流的元组并且产生新的数据流的元组，它们共同组成一个查询拓扑 Topology。查询拓扑在 Storm 中通过控制节点和工作者节点共同来处理。控制节点运行一个称之为 Nimbus 的后台程序，它类似于 Hadoop[24]的 JobTracker。Nimbus 负责在集群范围内分发代码、为 Worker 分配任务和故障监测。每个工作者节点运行一个称之 Supervisor 的后台程序。Supervisor 监听分配给它所在机器的工作，基于 Nimbus 分配给它的事情来决定启动或停止工作者进程。每个工作者进程执行一个查询拓扑的子集(也就是一个子查询拓扑)；一个运行中的查询拓扑由许多跨多个机器的工作者进程组成。由于一个完整的查询拓扑可能被分为多个子查询拓扑并由多个 Supervisor 完成，所以 Storm 依赖 Zookeeper[25]来管理 Nimbus 和多个 Supervisor 之间的协调工作。Zookeeper 是一个在分布式应用中通过维护配置信息以及分布式同步服务和组服务来提供可靠协调的节点。此外，Nimbus 后台程序和 Supervisor 后台程序都是快速失败和无状态的，所有状态维持在 Zookeeper 或本地磁盘。

S4[26]代表了一种通用开放的数据流解决方案，它能够提供并行分布式处理和故障容错的能力。S4 是基于 Java 语言开发的，类似于 Storm，S4 也依赖用户定义产生和处理数据流的函数。不仅如此，S4 也依赖 Zookeeper 来维护分布式集群的状态。S4 与 Storm 所共有的一个特征是它们都可以并行执行数据流算子，在 S4 中这被称为对称部署(Symmetrical Deployment)。然而，不同于 Storm 的是，S4 不提供动态负载均衡，而把这个工作交给了用户。S4 也提供了基于状态校验点的故障容错协议。在 S4 中，由于故障容错只依赖于状态校验点，因此可能导致(即便可能性很小)状态丢失。最近 S4Latin[27]则在 S4 的基础上实现了一个新的数据流处理框架，使得用户可以直接用类似查询的方式而不是编程的方式创建新的应用，这在很大程度上改善了 S4 平台的易用性。

Apache Samza[28]是 Linkedin 开发的并行分布式流处理系统，它是基于 Kafka 消息队列实现流数据处理的。Samza 使用 Apache Yarn[29]实现分布式资源分配和调度，同时使用 Apache Kafka[30]实现分布式消息管理。Samza 提供在 Yarn 管理的集群上创建和运行流处理任务的 API，在这个集群里，Samza 运行 Kafka 的代理，流处理任务负责处理 Kafka 产生的数据流，并且产生新的数据流。Kafka 提供了一个支持对消息持久化的分布式代理系统，该系统能够处理大规模消息并且具有一个消息持久化的文件系统。在 Kafka 中，一个数据流被称为一个 topic，这些 topic 通过划分被分布在不同的代理中，这种划分是通过与数据流中的消息关联的 key 来实现的，其中，具有相同 key 的消息被划分到一起。Samza 使用 Kafka 的 topic 划分来实现数据流的分布式划分。

Hadoop 已成为工业界和学术界进行云计算应用和研究的标准平台。目前，Hadoop 最初只用于进行批数据处理，因此当处理连续的数据流时，这种策略会导致工作节点具有较高的降速或者发生故障的可能性。为此，伯克利大学的 Tyson Condie 等人提出了一种改进型的 MapReduce 模型—HOP[31]，在该模型中，任务之间的中间数据通过管道连接，同时该模型也维持了之前 MapReduce 的编程接口以及故障容错机制。虽然管道连接对 MapReduce 模型提供了优势，但是也出现了一些挑战。首先，MapReduce 的故障容错机制是建立在中间状态被物化的基础上，为此，HOP 不得不在生产者周期性地把数据发送到消费者的过程中通过物化来支持故障容错机制。第二个挑战是管道的通信开销很大，这和 Combiners 函数所提供的批处理优化支持不一致，Combiners 函数通过在通信前预先执行聚集操作减少了通讯开销。最后，管道需要生产者和消费者能够智能地协同进行调度。针对上述挑战，HOP 给出了如下的管道设计策略：第一：HOP 用一个线程运行 Map 函数，并且把输出存放到缓存中，然后让另一个线程周期性地把缓存的内容发送到管道连接的 Reducer 端。第二：在缓存设计中，HOP 不用把缓存的内容直接发送到 Reducer 端，而是等到缓存的内容增长到一定的阈值，然后 Mapper 端执行 Combiner 函数，输出结果根据 partition 和 reduce key 进行排序，最后把缓存的内容以溢出文件的格式写入到硬盘中。第三：HOP 修改了 Hadoop 的 Job 提交接口使之能够接收 Job 列表，在这个列表中，每个 Job 都依赖它之前的 Job 并且通过它所依赖的 Job 标志符来标记。JobTracker 查找 Job 的标记，然后根据标记中的依赖关系协同调度 Job，相比下游的 Job，JobTracker 优先选择上游的 Job 进行调度。

然而，威斯康星大学的 Wang Lam 等人认为 MapReduce 或者它的变体不适合这类应用[32]，原因如下：首先，MapReduce 运行在一个数据集的静态 snapshot 上，而监控应用则需要处理不断变化的数据流。其次，每个 MapReduce 计算都有一个开始和结束，而数据流处理则永远不会结束。最后，当 MapReduce 的计算任务出现故障时，重新启动一个 MapReduce 的计算任务是可能的，但是对于数据流，这种可能性一般不存在。为此，他们提出了一种处理快速流动数据的框架—MapUpdate。类似 MapReduce，在 MapUpdate 中，开发人员只需要写入几个函数，特别是 Map 和 Update 函数，系统便能在集群中自动执行这些函数。但是 MapUpdate 在以下几个方面不同于 MapReduce。首先，MapUpdate 是在数据流上进行操作，所以 Map 和 Update 函数必须通过数据流来定义。第二，数据流永远不会终止，所以 Updaters 使用被称为 Slates 的存储器来记录它们在数据流上已经发现的概要数据。在 MapUpdate 中，Slates 实际上是 Updaters 的记忆内存，可以被分布在多个执行 Map/Update 的节点上，同时为了提供给将来的处理所使用，Slates 的数据也可以持久化到 Key-Value 数据库中。明确提出记忆内存 Slates，并且以几乎实时的方式像“上等公民”一样管理是 MapUpdate 的最重要特征。最后，大部分的 MapUpdate 应用不单是一个 Mapper 和一个 Updater，而是产生数据流和使用数据流的复杂的工作流。

加州大学的 MateiZaharia 等人提出了一种离散化数据流模型 D-streams[33]，其中核心思想是把一个数据流处理过程看成一系列在一个小的时间区域上确定性的批处理过程。D-Stream 模型的优点是一致性可以很好被定义(每条 Record 在它到达的时间间隔中被原子性处理)，并且这个处理模型很容易和批处理系统结合。除此之外，D-streams 能够在非常短的时间间隔内使用类似批处理系统的故障恢复机制，使之能够比现有 SPS 更加有效地减少出现故障的几率。

马德里理工大学的 Vincenzo Gulisano 等人提出了一种处理大规模数据的可扩展的并行分布式流处理系统—StreamCloud[34]，该系统使用了算子集并行化技术把一个查询划分为一些子查询，这些子查询通过被分配到不同的节点来减少总的查询执行时间。StreamCloud 的扩展协议是无侵入式的，能够根据输入负载有效地调整资源。此外，StreamCloud 通过把可扩展性和动态负载均衡相结合减少了总体使用的计算资源。

通过上述介绍，我们发现 SPS 在处理分布数据源时会面临以下三个方面的挑战。

第一个挑战是算子的并行化查询处理技术。这种并行化技术必须要保证语义的透明性，也即是由一个并行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。

第二个挑战是动态负载均衡和动态可扩展技术。现有的大部分 SPS 都是静态部署的，也就是说当 SPS 处理一个查询时，一旦这个查询(和算子)被部署后，它们就无法改变。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。正如前面介绍

的一样，数据流负载的波峰值和波谷值往往相差几个数量级，因此这种差异很可能会影响到并行分布式 SPS 的部署方案。也就是说，一个查询的静态部署方案可能无法适应当前的数据流负载。例如，当数据流的负载处在波峰时，已经分配的节点的数量可能比需要的要少，这被称为 under-provisioning，而当数据流的负载下降时，已经分配的节点的数量可能高于所需的节点的数量，这被称为 over-provisioning。值得注意的是，根据数据流负载的波动，无论是 under-provisioning 还是 over-provisioning，它们都会在不同的时刻影响查询的部署方案。为了克服这种缺陷，并行分布式 SPS 需要具有可扩展性，也就是说并行分布式 SPS 可以根据当前的数据流负载动态增加或者减少节点数量以实现在保证服务质量的前提下尽可能地减少使用的计算资源。为了实现这个目标，动态可扩展要与动态负载均衡相结合，这意味着，只有当已经分配的节点无法通过动态负载均衡来满足当前数据流负载对处理性能的要求时才需要增加新的节点。

第三个挑战是故障容错技术。随着节点数量以及它们之间通信的增多，并行分布式 SPS 出现故障的可能性也在不断提高，因此故障容错也就成为并行分布式 SPS 的一个基本要求。当前，很多故障容错技术已经在分布式 SPS 中被提出。然而，这些技术无法满足具有动态可扩展和动态负载均衡的并行分布式的 SPS 的要求。其中的原因包括两点，第一点是这些技术是为 SPS 的静态部署方案设计的，而并行分布式 SPS 的部署方案由于动态可扩展和动态负载均衡而不断变化。第二点是这些技术无法处理当并行分布式 SPS 由于动态可扩展或者动态负载均衡而重新部署时可能出现的故障，例如一个节点正在把其中的一部分负载交给较少负载的节点处理时，SPS 出现了一个故障。

3 并行分布式处理技术

下面我们分别介绍与上述三个挑战相关的技术现状。

3.1 并行化技术

S4 把数据流定义为一个形式为<K，V>的连续的事件元素集合，其中 K 和 V 分别是以元组形式存在的键和值。处理元素 PE 是 S4 中的基本计算单元。每个 PE 的实例都包括以下四个部分：(1)PE 定义的功能和相关配置，(2)输入的事件类型，(3)输入事件的键以及(4)输入事件的键值。每个 PE 实例都只是用它的键值对应的事件，因此如果没有对应事件键值的 PE 实例，那么 S4 会实例化这个 PE 实例，数据流根据事件的键值被 PE 实例所划分，如图 2 所示，MeasurementPE 是以 sliceID 为键，其中的两个实例分别对应 sliceID=”slice1”和 sliceID=”slice2”的事件。这种设计意味着 PE 可以分布在不同的节点上以达到并行分布式处理的效果。处理节点 PN 是 PE 的逻辑执行单元。它的功能包括监听事件，在输入事件上执行操作，在 PE 实例中分发事件以及发送输出事件。S4 利用键值的哈希函数把每个事件路由到 PN，一个事件可能被路由到多个 PN。所有可能的键值可以从 S4 的集群配置中获取。PN 中的事件监听器把输入事件发送到 PE 的容器，该容器负责以合适的顺序执行 PE 实例。

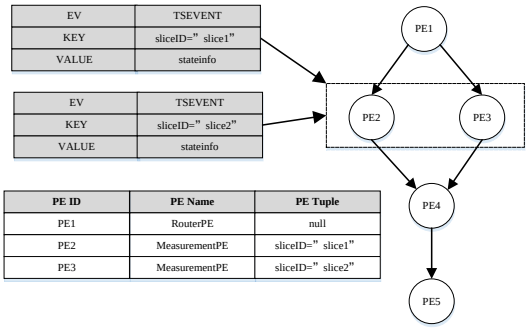

图 2 S4 的 PE 并行例子

Storm 把数据流中的每个元素定义为一个 tuple，一个 tuple 其实就是一个 value 列表，列表中的每个 value 都有一个 name，其中 value 可以是基本类型，字符类型和字节数组，当然也可以是其他可序列化的类型。

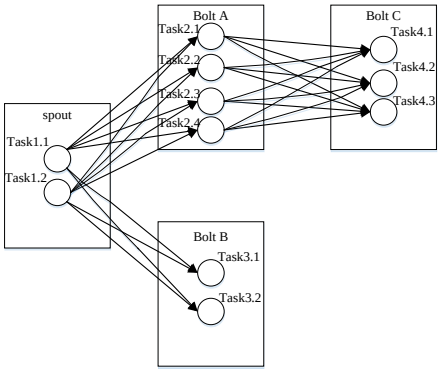

图 3 Storm 的 Bolt 并行例子

Storm 中，对于负责产生数据流的 Spout 和负责处理数据流的 Bolt 都有一个并行度，由用户在创建查询拓扑时指定。当查询拓扑提交到 Storm 后，Storm 会在集群内分配对应并行度个数的 Task 线程来并行执行它们，每个 Task 线程执行 Spout 或 Bolt 的一个实例。Task 之间通过流分组来表示它们的上下游关系，当上游的 Task 产生元组后，Storm 通过 ZeroMQ 把元组发送到订阅了该数据流的下游 Task。这个 ZeroMQ 是一个高性能异步的网络通信库，可以有效支持并行进程 Task 之间的通信[35]。如图 3 所示，Spout 并行度是 2，Bolt A、Bolt B 和 Bolt C 的并行度分别是 4、2 和 2，其中流划分方式是均匀分配，因此 Spout 或者 Bolt 中的每个 Task 线程的输出元组均匀分发到下一个 Bolt 的所有 Task 线程中。

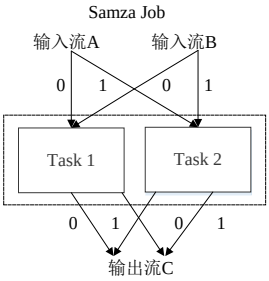

图 4 Samza 的 Job 并行例子

在 Samza 中，一个 Job 的基本处理流程是一个用户任务从一个或多个输入流中读取数据，再输出到一个或多个输出流中，具体映射到分布式消息管理 kafka 上就是从一个或多个 topic 读入数据，再写出到另一个或多个 topic 中去。多个 job 串联起来就完成了流式的数据处理流程。这种模式其实有点像 MapReduce 的过程，输入时由 kafka 的 partition 决定了分区和 task 数目，类似于一个 Map 过程，输出时由用户 task 指定 topic 和分区（或者框架自动由 Key 决定分区），这相当于一次 shuffle 的过程，下一个 job 读取新的 stream 时，可以认为是一个 reduce，也可以认为是下一个 map 过程的开始，如图 4 所示。不同之处在于 job 之间的串联无需等待上一个 job 的结束，实时的消息分发机制决定了整个串联的 job 是连续不间断的，亦即流式的。

然而，上面介绍的数据流处理任务没有区分有状态算子和无状态算子，为此 StreamCloud 提出了区分有状态算子和无状态算子的算子集并行化技术。为了给出它们的区别，下面以图 5(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(join 和 aggregate)以及 4 个无状态的算子(两个 map 和两个 filter)组成。这个查询部署在 90 个节点组成的集群上。在图 5(b)中，并行化单元是单个算子，每个算子都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 5(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(称为子查询)是有状态算子，每个查询被划分为有状态算子个数再加 1 个子查询。在图 5(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。

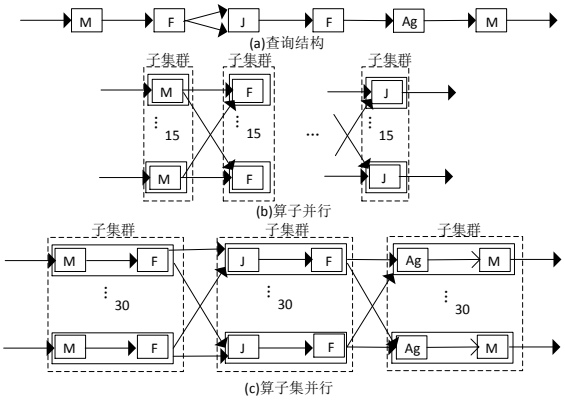

图 5 StreamCloud 的算子集并行例子

3.2　负载均衡技术

在数据流应用中，一些负载均衡协议已经被提出。其中最早的负载均衡协议是由 Aurora*提出的。这个协议被称为 Box Sliding 或者 Horizontal Load Sharing，是用来在运行时在节点之间移动算子。然而，这样的负载均衡解决方案不适合实际的应用，因为它只考虑了在拓扑两端移动算子的情况，而没有考虑如何通过在拓扑的中间位置移动算子来提高整个查询的吞吐量。如图 6 所示的 Box Sliding 的例子，在这个例子中，查询由 5 个算子组成，分别是 OP_A,…,OP_E，它们被部署到 3 个不同的节点，分别是 $Node_1$,…,$Node_3$。图 6a 描述了查询的初始部署：算子 OP_A 被部署到 $Node_1$，算子 OP_E 被部署到 $Node_3$，而其余的算子被部署到 $Node_2$，图 6b 描述了初始部署发生变化的情况，其中算子 OP_B 被移动到 $Node_1$，这种情况可能发生在当 OP_A 的选择性高于 OP_B 时(即 OP_A 输出流速率大于 OP_B 的输入流速率)。图 6c 描述了初始部署发生变化的另一种情况，其中算子 OP_C 和 OP_D 被移动到 $Node_3$，这种情况可能发生在 OP_C 和 OP_D 的选择性都低于 OP_B 的情况。值得注意的是，为了提高整个查询的吞吐量，查询部署的变更不单单只取决于算子的选择性，同时还要考虑其他因素，例如算子的计算成本，也就是说在图 6b 描述的例子中，只有 $Node_1$ 有足够的计算资源来保证两个算子的正常执行时，整个查询的吞吐量才会提高。

对于分布式流处理系统，当一个算子出现过载时，负载均衡协议就会在多个节点中重新分配这个算子的负载。例如文献[36]提出的动态负载均衡协议，其核心思想是在执行查询的多个节点上，对组成这个查询的每个算子部署多个实例，当其中一个算子实例的负载出现饱和时，动态负载均衡协议会将其负载分配给其

他闲置的算子实例。该协议并不需要任何集中式的组件来监控每个算子实例的状态。当一个算子实例的负载出现饱和时，这个算子实例会向上游节点发送“back pressure”消息，上游节点接收后开始把它的输出元组分配给这个过载算子的其他实例。然而，这个协议存在以下几点问题。首先，一个算子的多个实例被分配到多个节点上，意味着运行的算子实例必须和空闲的算子实例共享计算资源。其次，该协议只考虑无状态的算子，而改成适合有状态的算子则并不容易，其中在两个正在重新配置的算子实例之间需要一个状态迁移机制。

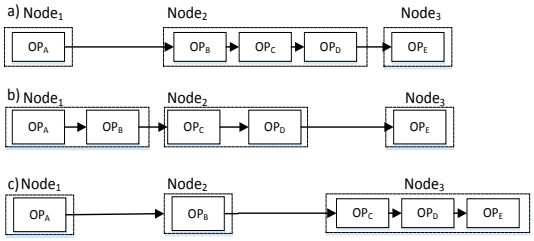

图 6 Box Sliding

在 S4 中，负载均衡是通过哈希函数静态地把事件路由到 PE 实例的方式实现的，这种方式无法适应动态的数据流负载，并且缺少健壮的在线 PE 迁移。和 S4 类似，Storm 也是通过用户在创建查询拓扑时静态设置的并行度来分配每个算子实例所处理的元组的，然而该并行度无法在运行时根据数据流的负载动态调整。在 Samza 中，负载均衡是通过 topic 划分和消息的关键字 key 来实现 Task 级负载均衡，而没有专门的在多个节点之间的负载均衡协议。在 StreamCloud 中，当运行的节点之间的处理负载出现不均时，负载均衡协议就会被动态触发。当已经分配的节点无法通过负载均衡处理输入负载时，StreamCloud 会自动提供新的节点，这被称为负载感知的供应，也即是根据当前负载和已经分配的节点个数来计算供应的节点个数。当输入负载可以通过更少的节点来管理时，多余的节点则被解除。

3.3 容错技术

在分布式数据库中，故障容错已经被广泛研究，并且出现了很多成熟的技术。就像 SPS 从数据库演化的一样，这些技术也在不断地被改进以满足新的需求。下面我们给出在 SPS 中提供故障容错的解决方案。这些解决方案可以看成是主动备份、被动备份和上游备份的修正和改进。

Storm 利用上游备份的变体来实现故障容错，在 Storm 中，spout 把元组保存到输出队列中直到这些元组被确认，这个确认发生在这个元组被拓扑成功处理后。如果一个元组在指定的时间内得到确认，那么该元组就被从输出队列中清除。然而，如果一个元组没有在指定的时间内，例如 30 秒内，得到确认，那么 spout 会重新发送这个元组并且再次经过拓扑的处理，这种通过拉的处理方式保证了元组至少被正确处理一次。对于 supervisor 节点，Nimbus 节点负责它们的异常处理。其中 supervisor 节点周期性地给 Nimbus 节点发送心跳，如果 Nimbus 没有在指定时间内接收到来自 supervisor 节点的心跳，那么 Nimbus 就认为这个 supervisor 节点已经失效并且把这个 supervisor 节点启动的 worker 移到其它的 supervisor 节点。由于消息失效的故障容错和节点失效的故障容错采用了两个没有关联的方式来处理，所以这种设计使得系统更加健壮。

在 S4 中，当一个节点出现故障后，S4 通过 ZooKeeper 来发现该故障并且把分配给出现故障节点的任务分配给其他的节点。当节点出现故障后，S4 无法保证消息一定被处理，其中 S4 使用 Gap 容错技术来实现恢复，处理节点的状态的快照定期被保存，以便当节点出现故障时创建具有相同状态的新的处理节点。由于 S4 的事件采用了推的发送方式，所以它会在高负载的情况下丢弃一些事件，这也导致 S4 无法保证每个事件被处理。

Samza 同样通过使用上游备份技术来处理消息处理故障，以保证消息至少得到一次正确处理。当一个节点出现故障时，新的节点会接管，这个新的节点开始从故障节点做的标记的上游的代理 topic 处读取数据。由于 Samza 对于确定性网络环境实现重复恢复，同时对于非确定性网络环境实现差异化恢复，所以上游备份恢复只对 task 级的故障起作用，如果一个代理出现故障，Samza 将丢失持久化到文件系统的消息并且无法恢复这些消息。

StreamCloud 在现有的故障容错技术上进行了以下几个方面的改进：(1)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；(2)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；(3)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[37]，减少了故障容错过程对运行时的影响。(4)基于 earliest timestamp 的容错协议可以对重新部署其间发生的故障进行容错。

D-Stream 把中间状态保存在可扩展分布式数据集 RDD 上进行故障容错，RDD 是一个无需副本、只需通过重新计算丢失数据的 lineage 就能够重构丢失数

据的存储抽象。为了快速地从故障中恢复，D-Stream 提供了故障节点状态的并行恢复。在集群中的每个节点都开始重新计算出现故障的节点的 RDD 的丢失部分，比同样无需副本的上游备份的恢复时间更少。由于对基本的复制操作都需要复杂的状态维护协议，所以并行恢复很难在 Record-at-a-time 系统中实现，但是 D-Stream 通过在大粒度的 RDD 分块上应用确定性的转换操作减少了维护成本。同时 D-Streams 采用了类似 MapReduce 的批处理框架，通过对进度慢的任务执行 Speculative 备份以在它们掉队后进行恢复。

4 未来挑战

尽管数据流处理技术在不断改进，但是随着大数据时代数据表现出越来越多的新特征，面向大数据的流处理技术仍将会面临很多挑战，下面归纳了研究人员和从业者未来不得不面对的其中几条：

处理架构：面向大数据的流处理架构未来可能需要同时支持历史数据和实时数据的处理，在这方面 Nathan Marz 等人提出了 Lamba 架构[38]，该架构被分解成批处理层，服务层和流处理层，解决了在任意数据上实时进行任意函数计算的问题，其中批处理层整合了 Hadoop，流处理层整合了 Storm。该系统的特点包括健壮、容错、可伸缩、通用，可扩展，支持 ad-hoc 查询以及维护成本低、可调试。

数据特征：在大数据时代，数据经常随着时间不断变化。因此面向大数据的流处理技术未来应该能够适应这个特征，并且能够在某些情况下发现数据的变化。

数据压缩：处理大数据，考虑存储数据的空间开销是非常重要的，这方面目前主要有两种方法：一种是无损压缩方法，另一种是选取一些具有代表性的数据采样方法。前者需要更多的时间开销，但是会减少空间开销，因此是一种时间换空间的方案。后者可能会丢失信息，但是会节省指数级的空间开销。因此面向大数据的流处理技术未来应该结合上述两种方法的优点设计专有的压缩技术。

数据发现：大数据时代，无标注的和无结构的数据正在变得越来越有用，2012 年的 IDC 关于大数据的研究报告[39]给出了这样一个分析结果，在 2012 年，数字世界中大约 23%(64 艾字节)的数据如果被标注或者解释，那么它们将会有用，然而，这些可能有用的数据目前只有 3%被标注。因此面向大数据的流处理未来应该设计针对这种数据的发现和处理技术。

5 总结

随着大数据时代到来，数据将会以前所未有的速度增长，这些数据表现的越来越多样、庞大和快速，如何处理这些数据成为大数据时代一个永恒的话题。为此，本文回顾了近期提出的面向大数据的流处理技术的现状，并且从流处理模型上对这些技术进行了划分，重点分析了面向大数据的并行分布式流处理模型的设计目标和架构，并且给出了未来的几个技术挑战，为大数据的研究人员和从业者带来一些思考和启示。

参 考 文 献

[1] Don Carney, Ugur Cetintemel, Mitch Cherniack et al.. Monitoring streams: a new class of data management applications[C]//In Proceedings of the 28th International conference on Very Large Data Bases, 2002: 215-226.

[2] Daniel J. Abadi, Don Carney, Ugur Cetintemel et al.. Aurora: a new model and architecture for data stream management[J]. VLDB Journal, 2003, 12(2):120-139.

[3] Arvind Arasu, Brian Babcock, Shivnath Babu et al.. Stream: The Stanford Data Stream Management System[J]. Springer, 2004, 1-21.

[4] Arvind Arasu, Shivnath Babu, Jennifer Widom. The CQL continuous query language: semantic foundations and query execution[J]. The VLDB Journal, 2006, 15(2):121-142.

[5] Hari Balakrishnan, Magdalena Balazinska, Don Carney et al.. Retrospective on aurora[J]. The VLDB Journal, 2004, 13(4):370-383.

[6]Sirish Chandrasekaran, Owen Cooper, Amol Deshpande et al.. TelegraphCQ: continuous dataflow processing[C]//In Proceedings of the 2003 ACM SIGMOD international conference on Management of data, 2003: 668.

[7] Amol Deshpande. An initial study of overheads of eddies[J]. SIGMOD Rec., 2004, 33(1):44-49.

[8] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska et al.. The design of the borealis stream processing engine[J]. In CIDR, 2005.

[9] Yanif Ahmad, Bradley Berg, Ugur Cetintemel et al.. Distributed operation in the borealis stream processing engine[C]// In Proceedings of the 2005 ACM SIGMOD international conference on Management of data, ACM, 2005: 882-884.

[10] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska et al.. Scalable distributed stream processing[C]// In CIDR, 2003.

[11] Stan ZdonikSbz, Stan Zdonik, Michael Stonebraker et al.. The aurora and medusa projects[J]. IEEE Data Engineering Bulletin, 2003, 51:3.

[12] M Tamer Özsu, Patrick Valduriez. Principles of Distributed Database Systems, Third Edition[M]. Springer, 2011.

[13] Nesime Tatbul, Ugur Çetintemel,Stan Zdonik et al.. Load shedding in a data stream manager[C]// In Proceedings of the 29th international conference on Very large data bases - Volume 29, 2003:309–320.

[14] Brain Babcock, Mayur Datar, Rajeev Motwani. Load shedding for aggregation queries over data streams[C]// In Data Engineering, 2004.

Proceedings. 20th International Conference on, 2004: 350-361.

[15] Yun Chi, Haixun Wang, Philip S. Yu. Loadstar: Load shedding in data stream mining[C]// In In Proc. Int. Conf. on Very Large Data Bases, VLDB'05, 2005: 1302-1305.

[16] Nesime Tatbul, Ugur Cetintemel, Stan Zdonik. Staying FIT: efficient load shedding techniques for distributed stream processing[C]// In Proceedings of the 33rd international conference on Very large data bases, 2007: 159-170.

[17] Brian Babcock, Shivnath Babu, Mayur Datar et al.. Models and issues in data stream systems[C]// In Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, 2002: 1-16.

[18] Mayur Datar, Rajeev Motwani. The sliding-window computation model and results[J]. Data Streams, 2007, 31:149.

[19] 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

[20] 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

[21] Gray, Jim and Graefe, Goetz. The five-minute rule ten years later, and other computer storage rules of thumb, ACM Sigmod Record, 1997, 26(4):63-68.

[22] Storm Project[EB/OL]. http://storm-project.net/.

[23] Jeffrey Dean, Sanjay Ghemawat. MapReduce: simplified data processing on large clusters[J]. Commun. ACM, 2008, 51(1):107-113.

[24] White Tom. Hadoop: The definitive guide[M], O'Reilly Media, Inc. 2012.

[25] Apache Zookeeper [EB/OL]. http://zookeeper. apache.org/.

[26] Yahoo S4 [EB/OL]. http://incubator.apache.org/s4.

[27] Paul Stoellberger. S4Latin: Language-Based Big Data Streaming, UK:Univeristy of Edinburgh,2011. [2012-10-02], http://analytical-labs.com/downloads/msc_BigDataStreams.pdf.

[28] Apache Software Foundation. Samza. [Online]. http://samza.incubator.apache.org/Apache Software Foundation. Apache Software Foundation.

[29] http://hadoop.apache.org/docs/current/hadoop--‐yarn/hadoop--‐yarn--‐site/YARN.html.

[30] Jay Neha Narkhede, Jun Rao Kreps. Kafka: A distributed messaging system for log processing, in In Proceedings of the NetDB, 2011.

[31] Tyson Condie, Neil Conway, Peter Alvaro et al.. MapReduce Online[C]// Proceedings of the 7th USENIX conference on Networked systems design and implementation, 2010: 21.

[32] Wang Lam, Lu Liu, STS Prasad et al..Muppet: MapReduce Style Processing of Fast Data[J], Proceedings of the VLDB Endowment, 2012, 5(12): 1814-1825.

[33] Matei Zaharia, Tathagata Das, Haoyuan Li et al.. Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters[C]// Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing, 2012:, 10.

[34] Gulisano Vincenzo, Jimenez-Peris Ricardo, Patino-Martınez Marta et al.. StreamCloud: An Elastic and Scalable Data Streaming System[J], IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, 2012, 23(12):2351-2365.

[35] ZeroMQ[EB/OL].http://zguide.zeromq.org/page:all

[36] Rebecca L. Collins, Luca P. Carloni. Flexible filters: load balancing through backpressure for stream programs[C]// In Proceedings of the seventh ACM international conference on Embedded software, 2009: 250-214.

[37] Zoe Sebepou, Kostas Magoutis. Scalable storage support for data stream processing[C]// In 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 2010: 1-6.

[38] N Marz, J Warren. Big Data: Principles and best practices of scalable realtime data systems. Manning Publications, 2013.

[39] J Gantz, D. Reinsel. IDC: The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East. December 2012.

张 鹏 男,1984 年生,博士后,主要研究领域为分布式系统和数据挖掘.Email:pengzhang@iie.ac.cn

李鹏霄 男,1986 年生,博士,工程师,主演研究领域为数据挖掘

任 彦 男,1978 年生,博士,高级工程师,主要研究领域为信息安全

林海伦 女,1987 年生,博士研究生,主要研究领域为数据挖掘和知识处理

杨 崃 男,1978 年生,博士,高级工程师,主要研究领域为网络安全

郑 超 男,1984 年生,博士,工程师,主要研究领域为数据挖掘和信息安全