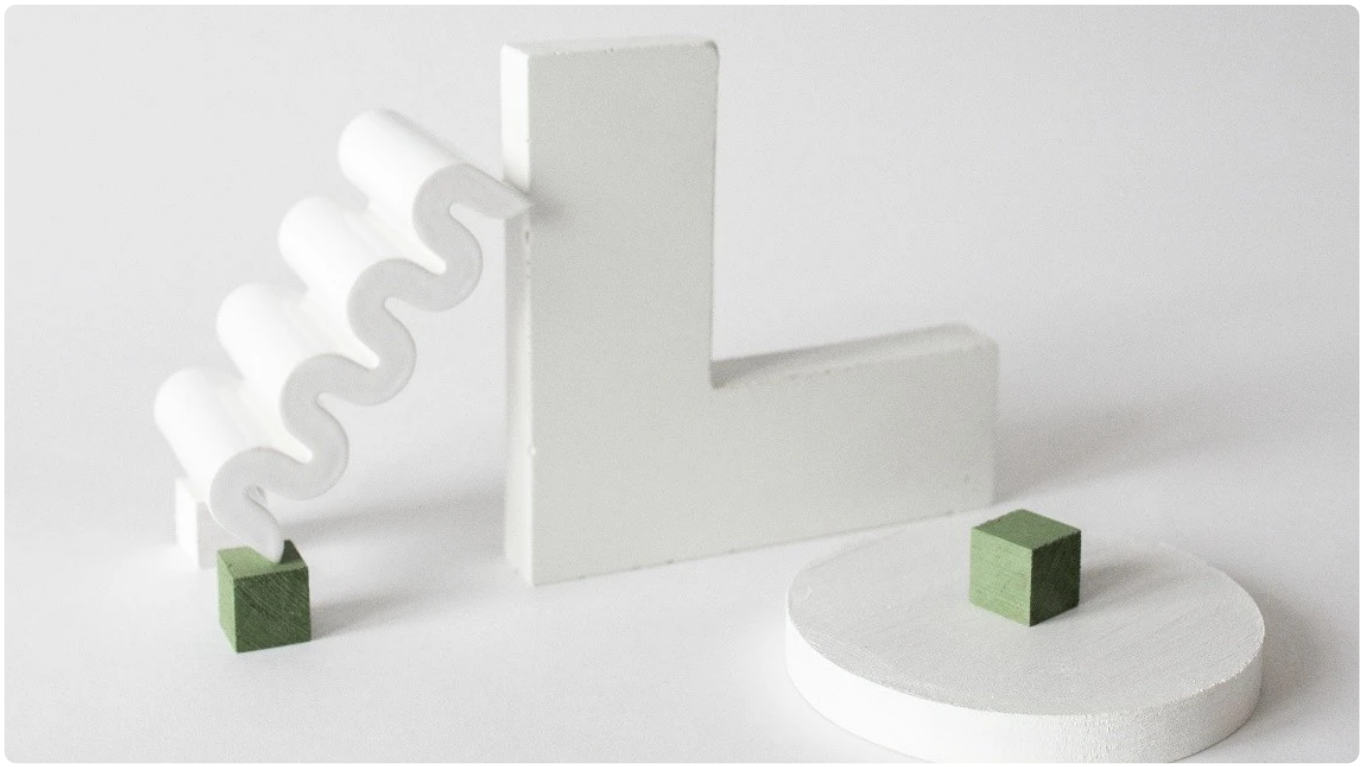
25 | 从S4到Storm（一）：当分布式遇上实时计算

2021-11-29 徐文浩

《大数据经典论文解读》 课程介绍 >



讲述：徐文浩
时长 17:40 大小 16.19M

你好，我是徐文浩。

到 Spanner 为止，我们已经把大数据里，关于数据存储和在线服务的重要论文解读完了。从这一讲开始，我们就要开始讲解另一个重要的主题，也就是大数据的流式处理。今天我们解读的第一篇论文，来自一个曾经辉煌但是今天已经逐渐销声匿迹的公司 Yahoo。这篇论文就是《S4:Distributed Stream Computing Platform》，伴随着这篇论文的，同样是一个开源系统 Apache S4。和同样孵化自 Yahoo 的 Hadoop 不同，S4 虽然是最早发布的开源分布式流式数据处理系统，但是在市场上最终却没有占有一席之地。

不过，学习 S4 的论文本身还是很有价值的。一方面，你可以看到大数据流式处理的基础结构是怎么样的，你会发现它和批处理的 MapReduce 是非常相似的；另一方面，它又会面临比批处理的 MapReduce 多得多的挑战和困难。

那在学完这一讲之后，你能有这样几点收获：

首先是对于大数据的流式计算问题的基本抽象，也就是它在逻辑上应该是一个什么样的模型，它想要解决的具体问题是什么。

其次是 S4 这个系统的设计是怎么样的，它是如何进行分布式计算的。

最后是 S4 系统的缺陷有哪些，以及有哪些问题 S4 干脆是回避了。这一点对于我们后面去认识 Storm、Kafka 这些系统的核心价值点非常关键。

好了，那接下来就请和我一起进入流式计算的世界吧。

实时计算到底有多“实时”？

在 MapReduce 出现之后，整个大数据处理领域就红火了起来，Hadoop 这样的开源项目也很快深入到各大互联网公司。一方面，MapReduce 帮我们解决了海量数据处理的问题；但是另一方面，我们常常又觉得 MapReduce 不太够用，因为很多时候，我们希望能够更加“实时”地处理数据。

之所以我们希望能够“实时”进行数据处理，是由于“效益”的原因。MapReduce 一开始的主要的应用领域，就是广告和搜索。通过 MapReduce 程序，我们可以分析海量的用户搜索行为、广告点击行为，来帮助我们优化搜索排序和广告展示。

比如在广告领域，最简单的一个优化的办法，就是统计新广告的点击率，点击率太低的广告，我们不予展示，把展示机会给别的广告就好了。在论文里，Yahoo 也举了这样一个例子，通过在线统计点击率，把低质量的广告过滤掉，S4 可以帮助他们再提升 3% 的点击率，并且对广告收入没有任何影响。

这个需求，和我们通过 MapReduce 生成各种报表是一回事儿。唯一的一个差异在于，**我们希望这个数据统计的反馈的时间能够尽量短一些，也就是“实时一点”。**而这个差异，就触到了 MapReduce 的痛点了。

一般来说，我们的 MapReduce 都是定时执行的，比如每天运行一次，生成一个报表，或者频繁一点，每小时运行一次，计算上一个小时的点击率数据。但是，这个获得反馈数据的频率还是太慢了。每小时运行一次 MapReduce 程序，意味着我们的统计数据，平均要晚上半个小时。而半个小时里，低质量的广告或者搜索结果已经曝光了很多次了。

举个例子，现在无论是什么样的社会热点新闻，很容易在微博热搜上出现。往往在新闻发生后的一两分钟，就已经有大量的搜索出现在微博热搜里了。如果我们要等待半个小时，才能统计到这些搜索，那么热搜功能就可以说形同虚设了。

所以，我们希望能在尽可能短的时间内，就得到这样的反馈数据。那么，你可能要问了，我们能不能直接更频繁地运行 MapReduce 程序呢？比如，每分钟运行一次，是不是就解决这个问题了？

如果你仔细学习了之前的课程，相信你自己也会意识到这样是行不通的。采用频繁运行 MapReduce 程序的办法，我们至少会遇到两个问题。

一个是大量的“额外开销”。

我们之前讲过，MapReduce 的额外开销不小，再小的任务也需要个十几秒到一分钟的运行时间。如果我们高频率每分钟运行 MapReduce 任务，那么“额外开销”占的时间比重和硬件资源会非常高，也很浪费。

二个是我们不得不让输入文件变得极其“碎片化”。

无论是 GFS 还是 HDFS，都是把文件变成一个个 64MB 大小的 Block，然后 MapReduce 通过分布式并行读取来进行快速分析。

但是如果我们需要每分钟都处理数据，那么对于输入的数据，就要按照分钟进行分割。每分钟我们都需要有很多个文件，分布到 GFS/HDFS 上不同的数据节点。这样，我们的文件都会变得很小，也就丧失了顺序读取大文件的性能优势。

其实，高频率地执行 MapReduce 还会有很多问题。而归根到底，是这两点：

第一，**MapReduce 是为“高吞吐量”而设计的一个系统**。在整个系统设计理念里，它没有考虑“低延时”这个需求。

第二，**MapReduce 的数据，是一份“边界明确（bounded）”的数据**。在进行数据处理之前，要处理的数据已经存放在存储系统上了。而我们想要进行的实时数据统计，想要处理的是一份“无边界（unbounded）”的数据，会不断地有新数据流入进来，永无停歇。

所以，我们需要一个全新的流式数据处理系统，这也是 S4 这个系统的出发点。

流式计算的逻辑模型

我们先来看一看 S4，是怎么抽象我们的流式计算的。S4 把所有的计算过程，都变成了一个个**处理元素**（Processing Element）对象，简称为 PE 对象。我这里特地加上了对象，就是因为在实现上，PE 就是一个面向对象编程里面一个实际的对象。

每一个 PE 对象，都有四部分要素组成，分别是：

PE 本身的**功能（functionality）**，这个体现为 PE 类里实现的业务逻辑函数，以及为了这个类配置的各种参数；

PE 能够处理的**事件类型（types of events）**；

PE 能够处理的事件的**键（keyed attribute）**；

PE 处理的事件的键对应的**值（value）。**

对于流式的数据处理，就是由一个个 PE 组成的有向无环图（DAG）。有向无环图的起点，是一些特殊的被称为**无键 PE**（Keyless PE）的对象。这些对象的作用，其实就是接收外部发送来的事件流，这些外部发送过来的事件流，其实就是一条条的消息。

这些无键 PE 会解析对应的消息，变成一个个事件。然后给每个事件打上三个信息，分别是：

事件类型（Event Type）；

事件的 Key；

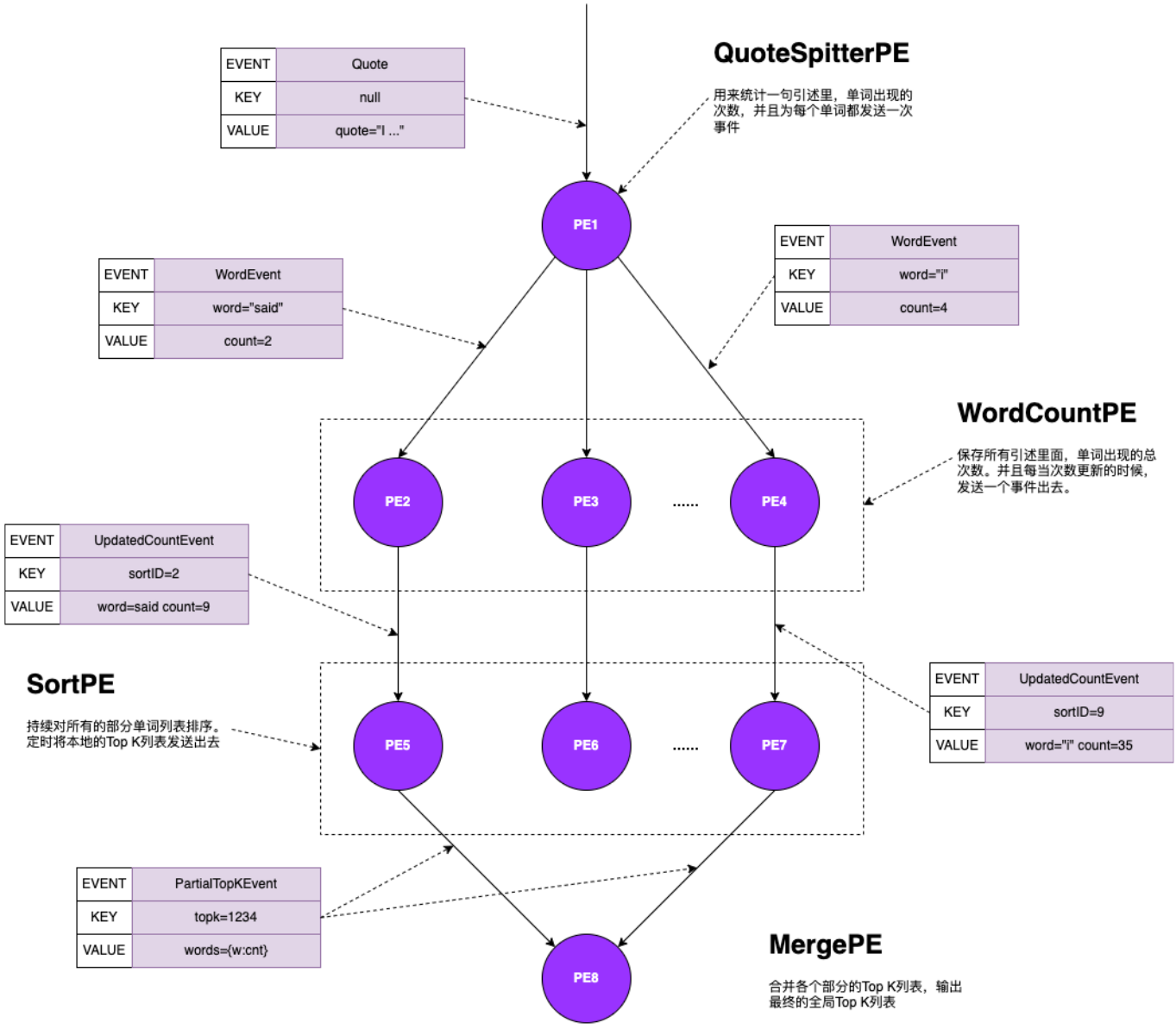
事件的 Value。

然后可以把事件给发送出去。接着下游的其他 PE 对象，会根据自己定义的事件类型，和能处理的键来接收对应的消息，并且处理这个消息。如果当前系统里，没有对应的键的 PE，那么系统会创建一个新的 PE 对象。

处理数据的 PE 对象，可以选择处理完之后立刻发送一个新的事件出去；也可以选择在对象内部来维护一个状态，然后当处理了一定数量的消息之后，或者过了一个固定的事件间隔

之后把消息发送出去。

最后，在整个有向无环图的终点，会有一系列的 PE 对象。这些对象，会把最终的计算结果**发布（Publish）。**这个发布的频率，也和其他 PE 发送消息的逻辑类似，可以在每收到一个事件就发送，也可以要求接收到一定数量的事件，或者每隔一个特定的时间间隔发送。



论文中的图1，求出现的单词次数Top K的例子

这么来描述，可能整个过程有点过于抽象，我们还是来一起看一看论文里图一的示例。这个例子，是用来统计整个系统里，出现得最多的 K 个单词，也就是 Top K，它的整个 DAG 的结构是这样的：

首先，在 DAG 的起始节点，是一个 **QuoteSplitterPE**，这个 PE 也是一个无键 PE。

- 它负责接收外部发送来的句子，然后分割成一个个单词，接着会统计单词在句子里面出现的次数。
- 然后，这个 PE 会把每个单词的出现次数，作为一个 WordEvent 发送出去。对应的 Event 的 Key 就是（Word，具体单词）的这么一个组合（Tuple）。而对应的 Event 的值，就是（Count，出现次数）的这么一个组合（Tuple）。

第二层里，是一系列叫做 **WordCountPE** 的 PE 对象。它在系统里面申明，我只接收 WordEvent。然后每个不同的单词，都会有一个对应的 PE 对象。所以可以想象，整个系统中会有海量的 PE 对象。

- 它的逻辑也很简单，上游的 PE，会把相同单词的 WordEvent 都发送到同一个 PE，那么这个 PE 里，就可以统计到这个单词出现的总的次数。
- 每当收到一个事件，这个单词的出现次数就会更新，对应的它就会向下游，发送一个 UpdatedCountEvent，也就是更新单词计数的事件。
- 这个事件里，对应的 Key 是（Sort，N）这样一个组合，每一个 PE 对象里的 N 都是随机的，但是固定不变的。这个组合是为了下一层的负载均衡，我们可以自己去设定 N 这个参数，N 越大，意味着下游的 PE 对象越多，负载就会分配到更多不同的对象里去计算的。而对应的值，则包括了对应的单词是什么，以及对应的单词的出现次数。也就是 ((Word，具体单词), (Count，出现次数)）这么一个组合。

第三层里，则是一系列叫做 **SortPE** 的对象。它的作用则是接收上游不同单词的出现次数，然后在内部进行排序。最后输出自己内部排序的 Top K，再给到下游。本质上，它相当于是所有单词的某一个分区的数据。这个分区，包含了一部分单词的所有数据。我们前面设定了 N 是几，我们就会有几个 SortPE 的对象。

- 给到下游的事件，叫做 PartialTopkEvent，看名字你就知道它包含的信息，就是一个部分数据的 Top K。
- 所有 SortPE 的对象，输出的消息的 Key 都是相同的，因为为了获得全局的排序，它们需要发送给同一个 PE 对象。在这里，这个 Key 就被写死成了（topk，1234） 这么一个组合。而 Value，则是 K 个（单词，出现次数）的集合。

而整个 DAG 的终点，则是唯一的一个 **MergePE**。它的作用，就是接收 PartialTopkEvent，然后在内部进行一次归并，选出全局的 TopK。并且最终，它还需要把对应的数据，写入到外部其他的存储系统，比如数据库里，供其他的应用读取。

S4 这个把整个数据处理流程，变成一个有向无环图的设计，也是后续所有流式处理系统都采用的一个解决方案。所有的数据，变成了事件流，而开发人员只需要做两件事情：

第一，是设计整个 DAG 应该是什么样子的。

第二，是实现这个当中每一个节点的业务逻辑代码。

而开发人员，不需要关心数据是在哪里被处理的。这些，都由 S4 这个分布式系统自己来决定。

师从 MapReduce 的设计理念

其实 S4 的系统架构，和我们之前看过的 MapReduce 这样的框架一脉相承。PE 其实和 Map/Reduce 函数一样，只是一个抽象的概念。不过 S4 的系统设计，要更加激进一点，那就是 **S4 选择了一个无中心的，完全对称的架构**。

S4 和我们之前看过的所有系统都不一样，没有所谓的 Master 节点。如果一定要说有一个中心化的地方的话，S4 依赖于 Zookeeper，也就是一个类似于 Chubby 这样的分布式锁系统。S4 的所有服务器，都会作为一个**处理节点**（ProcessingNode），简称 PN 注册在 Zookeeper 上。具体如何分配负载，是**由各个节点协商决定的，而不是由一个中心化的 Master 统一分配**。

每一个处理节点，都是相同的，它由上下两部分组成。

上面，是实际的业务处理逻辑模块：

它通过 Event Listener，监听外部发送过来的消息，转发给对应的 PE 对象。

PE 对象的所有输出结果，都发送给 Dispatcher，让 Dispatcher 确定应该发送给哪些 PE 里。

实际的消息发送，会由 Dispatcher 交给 Emitter，对外发送出去。

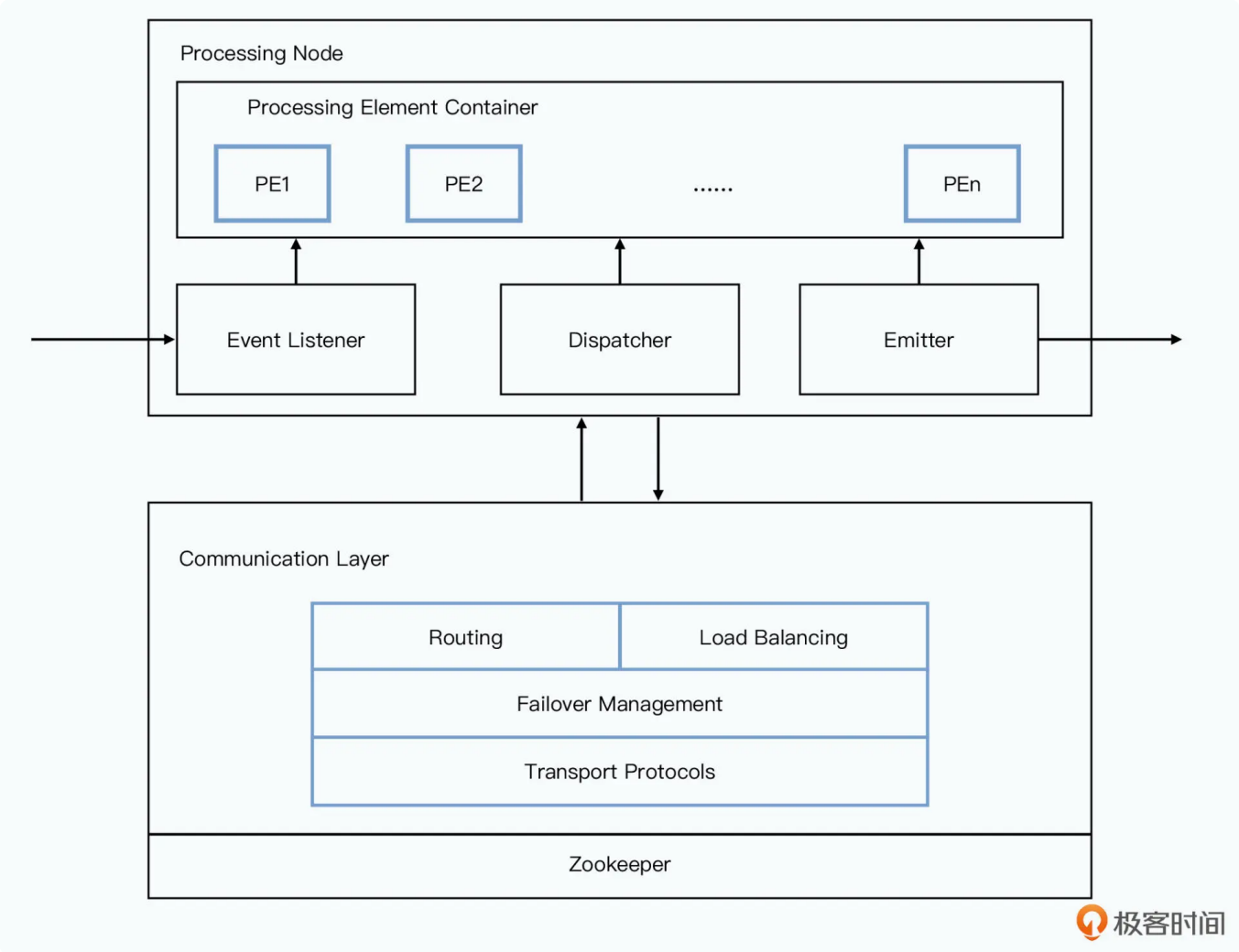
业务处理模块里，只会确定对应的消息发送，应该发送给哪一个逻辑上的 PE，实际具体发送到哪一台物理节点，则是由**下面的通信层模块**来决定的。这个模块主要解决这样几个问题：

首先是具体的**路由**，也就是 Event 要去的某一个逻辑 PE，到底在哪台物理服务器上，是由通信层模块来找到并且发送出去的，上层的业务处理流程不需要知道。

其次是**负载均衡**，不同的单词，更新的频率可能不一样。所以不同的处理节点的负载也会不一样。当有一个新的单词出现的时候，我们需要判断新的 PE 应该放到哪一个节点上去。

然后就是底层的**容错恢复机制**了，当有特定节点挂掉的时候，我们需要在其他的节点上，恢复原先这个节点被分配的 PE。

最后就是实际的**传输协议**，S4 是一个“插件式”的架构，也就是底层的传输协议也是可以切换的。S4 既支持通过 TCP 发送消息，确保消息能够发送成功，也支持通过 UDP 发送消息，来支持更大的吞吐量。



论文中的图2，单个S4节点的架构

你可以看到，这个其实和我们看 MapReduce 的框架是类似的，开发人员的关注点，只需要在 PE 这个纯粹的业务逻辑层面。至于计算在哪一台服务器上发生，各个节点之间是怎么通信的，开发人员完全不需要关心。

稍显“过时”的伸缩和容错能力

不过，看到这里，相信你也会有一些疑问。以单词计数为例，看起来一个 S4 在线上的有向无环图就需要有海量的对象，这个数量级可能是数万乃至数十万。而不像之前我们看过的 MapReduce 那样，只需要有少数 Map 和 Reduce 就好了。

没错，S4 的设计其实还有粗糙，也还有着很多的问题。

首先就是这里的**海量对象的问题**。由于每一个处理数据的 Key 都要是一个对象，系统里就会有海量的对象。而一个 Key 如果只出现一次，之后再也不出现了，也要占用内存。S4 对此的解决办法，是给 Key 设定 TTL，定期清理掉不需要的 Key。

其次，是 **S4 里，没有时间窗口的概念**。在我们进行实时数据处理的时候，我们需要统计的，常常是“过去一分钟的热搜”，或者“过去一小时的热搜”，这样有一个时间范围的数据。

但是在 S4 的设计里，我们并没有地方可以设定这个时间窗口。所以类似的需求，需要我们自己在 PE 的代码里面去维护或者实现，一下子大大增加了开发的难度和复杂度。

第三，是 **S4 的容错处理非常简单**。S4 能够做到的容错，其实就是某一个计算节点挂掉了，我们重新再起一个计算节点承担它的工作。但是，原先节点里，所有 PE 维护的状态信息就都丢失了。我们既不知道目前的统计信息是什么，也不知道目前处理到哪些事件了。

Yahoo 给出的答案是退回到离线批处理计算的数据上，但是这个显然就不满足流式处理一开始的需求了。只能算是个聊胜于无的方案。

最后一个问题，则是 S4 虽然是一个分布式系统，但是**并不支持真正的动态扩容**。在一开始论文的假设部分，就假设了运行中的集群不会增加或者减少节点。

这样带来的问题，就是当负载快速上升的时候，S4 的策略是随机丢弃一些数据，本质上是对数据进行了采样，而不是能够通过简单增加硬件来解决问题。

不过不管怎么说，S4 还是让大数据的的流式处理迈出了第一步。而这些 S4 并没有回答好的问题，也会为接下来的流式数据处理系统的兴旺拉开了帷幕。

小结

好了，对于 S4 的论文，我们到这里也就解读完了。我们看到，随着大数据的价值深入人心，MapReduce 这样定期定时进行数据处理的方式，逐渐难以满足业务需求。于是，大数据的流式计算登上了历史舞台。

Yahoo 通过 S4 系统，进行**低延时的“实时”数据处理**。整个系统的设计理念类似于 MapReduce，开发人员只需要实现 Processing Element 这样的业务处理逻辑，而不需要关心“分布式”是怎么运行的。整个框架会完成数据的分发、计算节点的调度，以及容错之后的恢复。

通过 S4，Yahoo 能够及时地获取广告和搜索数据的反馈，以及进行在线的 A/B 测试。整个 S4 内部的设计，也将业务逻辑层和网络协议、数据路由、负载均衡等拆分开来，做成了一个**可插拔**（Pluggable）的系统架构。

在整个的流式数据处理框架里，S4 采用了一个典型的 **Actor 模式**。整个数据处理的流程，可以被画成一个有向无环图，图里的每一个点都是一个处理元素，每一条边都是一条消息传递的路径，而每一个处理元素都会被托管在某一个处理节点里。

处理元素负责实现业务逻辑，并且可以保存计算结果在内存。同时，S4 支持你定时地将对应的结果**发布**到外部的存储系统里，使得计算结果对外可用。

但是，S4 的设计显然也是很粗糙的。**S4 采用了一个完全对称、没有中心节点的分布式架构，虽然看起来这个解决了“单点故障”问题，但是也因此放弃了动态扩容**，而只能在大量流量进入的时候，选择服务降级的解决方案。

而在业务层面，S4 的容错，也**只是考虑“计算节点”层面的容错**。容错只是将挂掉的节点能够在其他的硬件上重新运行起来，但是已经处理的历史数据都已经丢失了。而**对于节点之间的数据传输，S4 也没有作出全链路的传输保障**。

这些问题，也是后面的 Storm、Kafka、Flink 这些系统出现的出发点。而 S4 自己，却在 2014 年就从 Apache 孵化的项目中“退役（retried）”了。

推荐阅读

S4 的论文是流式计算的起点，所有 S4 设计上的各种缺陷，都成为后来的系统的改进点。而它的论文本身也非常简短，读一读论文原文非常有助于你认识到流式计算最原始和粗糙的想法，是从哪里开始的。

思考题

除了我在这节课里所说的问题之外，每分钟定时运行 MapReduce 来进行实时数据处理，还可能会遇到哪些问题呢？我们是否可以通过优化改造 MapReduce，来解决这些问题呢？请你想一想，在留言区中分享你的思考和答案。

上一篇 复习课（六）| Hive

下一篇 26 | 从S4到Storm（二）：位运算是个好东西

精选留言 (2)

写留言

在路上

2021-11-29

徐老师好，第一、MapReduce的每个Reduce任务都会对数据排序，哪怕需要的顺序和输入的顺序一致，在运算大数据集时排序可以让相关的数据靠在一起，减少内存的使用，但是对流式计算来说，把一分钟的数据全部放入内存是可行的，排序应该只在需要的时候进行。

...

展开

3

陈迪

2021-12-01

communication layer 类似于node的sidecar😄

展开