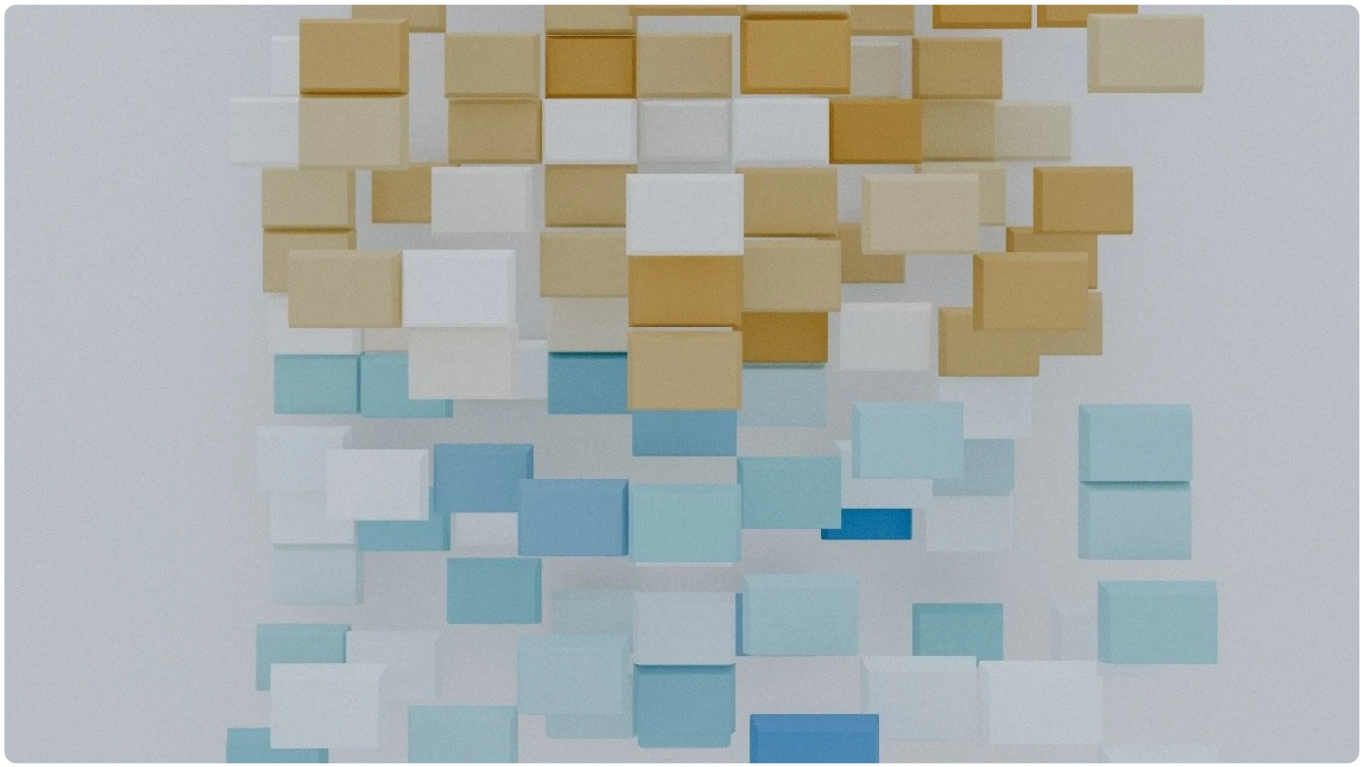
28 | Kafka（二）：从Lambda到Kappa，流批一体计算的起源

2021-12-10 徐文浩

《大数据经典论文解读》 课程介绍 >



讲述：王惠
时长 20:59 大小 19.23M

你好，我是徐文浩。

在上节课里，我们已经了解了 Kafka 的基本架构。不过，对于基于 Kafka 的流式数据处理，我们还有两个重要的问题没有回答：

> 第一个，Kafka 的分布式是如何实现的呢？我们已经看到了 Kafka 会对数据进行分区，以进行水平扩展。那么，如果我们可以动态添加 Broker 来增加 Kafka 集群的吞吐量，这个集群的上下游是怎么知道的呢？
>
> 第二个，在我们有了 Kafka 和 Storm 这样的系统之后，我们的流式处理系统应该怎么搭建呢？我们如何解决可能遇到的各种故障带来的数据不准确的问题呢？

那么，今天这节课，就是要帮助我们回答这两个问题。**一方面**，今天我们会深入来看一下，Kafka 是如何随着 Broker 的增加和减少，协调上下游的 Producer 和 Consumer 去收发消息的。**另一方面**，我们会从整个大数据系统的全局视角，来看一下在有了 Kafka 和 Storm 这样的利器之后，我们的大数据系统的整体架构应该如何搭建。

Kafka 的分布式系统的实现

首先，Kafka 系统并没有一个 Master 节点。不过，这一点倒是不让人意外，主要是 Kafka 的整体架构实在太简单了。我们在上一讲就看到了，单个的 Broker 节点，底层就是一堆顺序读写的文件。而要能够分布式地分摊压力，**只需要用好 ZooKeeper** 就好了。

每一个 Kafka 的 Broker 启动的时候，就会把自己注册到 ZooKeeper 上，注册信息自然是 Broker 的主机名和端口。在 ZooKeeper 上，Kafka 还会记录，这个 Broker 里包含了哪些主题（Topic）和哪些分区（Partition）。

而 ZooKeeper 本身提供的接口，则和我们之前讲解过的 Chubby 类似，是一个分布式锁。每一个 Kafka 的 Broker 都会把自己的信息像一个文件一样，写在一个 ZooKeeper 的目录下。另外 ZooKeeper 本身，也提供了一个监听 - 通知的机制。

上游的 Producer 只需要监听 Brokers 的目录，就能知道下游有哪些 Broker。那么，无论是随机发送，还是根据消息中的某些字段进行分区，上游都可以很容易地把消息发送到某一个 Broker 里。当然，Producer 也可以无需关心 ZooKeeper，而是直接把消息发送给一个负载均衡，由它去向下游的 Broker 进行数据分发。

高可用机制

而在 Kafka 最初的论文里，还没有包括 Kafka 的高可用机制。在这种情况下，一旦某个 Broker 节点挂了，它就会从 ZooKeeper 上消失，对应的分区也就不见了，自然数据我们也就没有办法访问了。

不过，在了解了这么多的分布式高可用方案之后，相信我们要自己实现一个 Kafka 的高可用方案，自然也不困难。在 Kafka 发布了 0.8 版本之后，它就支持了由多副本带来的高可用功能。

在现实中，Kafka 是这么做的：

首先，为了让 Kafka 能够高可用，我们需要对于**每一个分区都有多个副本**，和 GFS 一样，Kafka 的默认参数选择了 3 个副本。

其次，这些副本中，有一个副本是 Leader，其余的副本是 Follower。我们的 Producer 写入数据的时候，**只需要往 Leader 写入**就好了。Leader 自然也就是将对应的数据，写入到本地的日志文件里。

然后，每一个 Follower 都会从 Leader 去**拉取最新的数据**，一旦 Follower 拉到数据之后，会向 Leader 发送一个 Ack 的消息。

我们可以设定，有多少个 Follower 成功拉取数据之后，就能认为 Producer 写入完成了。这个可以通过在发送的消息里，设定一个 acks 的字段来决定。如果 acks=0，那就是 Producer 的消息发送到 Broker 之后，不管数据是否刷新到本地硬盘，我们都认为写入已经完成了；而如果设定 acks=2，意味着除了 Leader 之外，至少还有一个 Follower 也把数据写入完成，并且返回 Leader 一个 Ack 消息之后，消息才写入完成。我们可以**通过调整 acks 这个参数，来在数据的可用性和性能之间取得一个平衡**。

Producer 发送数据、Broker 接收并存储下来的逻辑就这么简单。不过，下游 Consumer 去消费数据的逻辑就稍微复杂一点了。主要的挑战，来自于我们可以动态增减 Broker 和 Consumer。

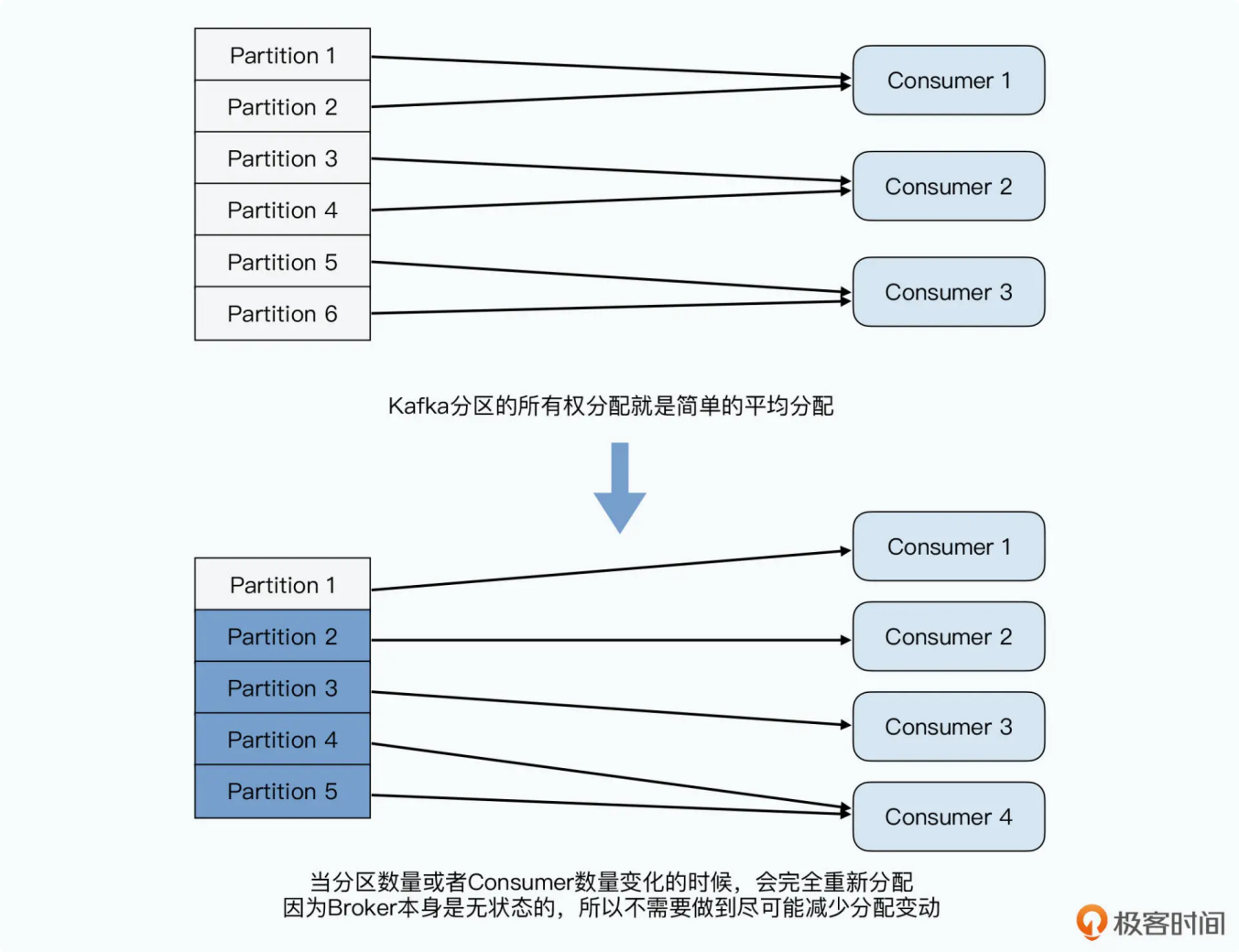
负载均衡机制

Kafka 的 Consumer 一样会把自己“注册”到 ZooKeeper 上。在同一个 Consumer Group 下，一个 Partition 只会被一个 Consumer 消费，这个 Partition 和 Consumer 的映射关系，也会被记录在 ZooKeeper 里。这部分信息，被称之为“**所有权注册表**”。

而 Consumer 会不断处理 Partition 的数据，一旦某一段的数据被处理完了，对应这个 Partition 被处理到了哪个 Offset 的位置，也会被记录到 ZooKeeper 上。这样，即使我们的 Consumer 挂掉，由别的 Consumer 来接手后续的消息处理，它也可以知道从哪里做起。

那么在这个机制下，一旦我们针对 Broker 或者 Consumer 进行增减，Kafka 就会做一次数据“再平衡（Rebalance）”。所谓再平衡，就是**把分区重新按照 Consumer 的数量进行分配，确保下游的负载是平均的**。Kafka 的算法也非常简单，就是每当有 Broker 或者 Consumer 的数量发生变化的时候，会再平均分配一次。

如果我们有 X 个分区和 Y 个 Consumer，那么 Kafka 会计算出 N=X/Y，然后把 0 到 N-1 的分区分配给第一个 Consumer，N 到 2N-1 的分配给第二个 Consumer，依此类推。而因为之前 Partition 的数据处理到了哪个 Offset 是有记录的，所以新的 Consumer 很容易就能知道从哪里开始处理消息。



Kafka的分区分配策略非常简单，就是简单的平均分配

而和 Storm 一样，本质上，**Kafka 对于消息的处理也是“至少一次”的**。如果消息成功处理完了，那么我们会通过更新 ZooKeeper 上记录的 Offset，来确认这一点。而如果在消息处理的过程中，Consumer 出现了任何故障，我们都需要从上一个 Offset 重新开始处理。这样，我们自然也就避免不了重复处理消息。

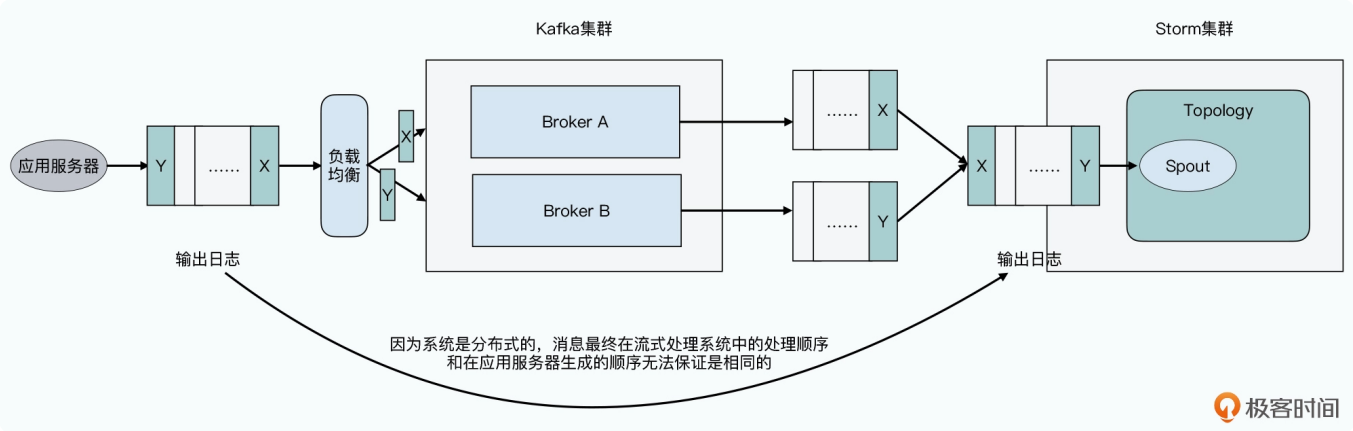
如果你希望能够避免这一点，你需要在实际的消息体内，有类似 message-id 这样的字段，并且要通过其他的去重机制来解决，但是这并不容易做到。

顺序保障机制

不过，Kafka 虽然有很强的性能，也在发布之后很快提供了基于多副本的高可用机制。但是 Kafka 本身，其实也是有很多限制的。

首先，是 **Kafka 很难提供针对单条消息的事务机制**。因为我们在 ZooKeeper 上保存的，是最新处理完的消息的一个 Offset，而不是哪些消息被处理完了、哪些消息没有被处理完这样的 message-id => status 的映射关系。所以，Consumer 没法说，我有一条新消息已经处理完了，但是还有一条旧消息还在处理中。而是只能按照消息在 Partition 中的偏移量，来顺序处理。

其次，是 **Kafka 里，对于消息是没有严格的“顺序”定义的**。也就是我们无法保障，先从应用服务器发送出来的消息，会先被处理。因为下游是一个分布式的集群，所以先发送的消息 X 可能被负载均衡发送到 Broker A，后发送的消息反而被负载均衡发送到 Broker B。但是 Broker B 里的数据，可能会被下游的 Consumer 先处理，而 Broker A 里的数据后被处理。



因为分布式的存在，应用服务器发出的消息，在实际处理的时候，顺序可能会发生变化

不过，对于快速统计实时的搜索点击率这样的统计分析类的需求来说，这些问题都不是问题。而 Kafka 的应用场景也主要在这里，而不是用来作为传统的消息队列，完成业务系统之间的异步通信。

数据处理的 Lambda 架构

其实，有了 Storm 和 Kafka 这样的实时数据处理架构之后，另一个问题也就浮出了水面。**既然我们已经可以获得分钟级别的统计数据，那我们还需要 MapReduce 这样的批处理程序吗？**

答案当然还是需要的，因为在目前的框架下，我们的流式计算，还有几个问题没有处理好。

首先，是我们的**流式数据处理只能保障“至少一次（At Least Once）”的数据处理模式**，而在批处理下，我们做到的是“正好一次（Exactly Once）”。也就意味着，批处理计算出来的数据是准确的，而流式处理计算的结果是有误差的。

其次，是当数据处理程序需要做修改的时候，**批处理程序很容易修改，而流式处理程序则没有那么容易**。比如，增加一些数据分析的维度和指标。原先我们只计算点击率，现在可能还需要计算转化率；原先我们只需要有分国家的统计数据，现在还要有分省份和分城市的数据。

我们原先的计算结果已经保存在数据库或者 HDFS 上了。那么对于批处理程序来说，我们的解决方案也很容易，那就是选定一个我们希望新的报表需要覆盖的**时间范围**，比如过去 30 天。我们撰写一个新的 MapReduce 程序，运行出新的计算结果，保存成新的数据表。我们可以把旧的数据表删除，用新的数据表替换就好了。

通常，我们的 Hadoop 集群不只要承担报表任务，也会承担很多临时的分析任务。所以一般来说，像 Hadoop 这样的批处理集群的计算资源对于单个报表来说是足够富余的，重跑 30 天的数据分析，往往也可以在 1~2 天内完成。

流式数据处理的性能压力

但是对于流式处理，问题就有些麻烦了，特别是在没有 Kafka 的时候。

我们重新撰写一个新的 Storm 的 Topology，来支持新的分析维度和指标并不困难。**困难的地方在于，我们需要在不影响正在线上运行的程序的情况下，进行新版本程序的发布。**

一个解决方案是，我们写了一个新的 Topology，然后需要**重放**（Replay）过去 30 天的日志数据。如果我们用的是 Scribe 这样的流式日志传输系统，我们会发现日志流都已经上传到了 HDFS 上，我们还需要有一个程序，从 HDFS 上把数据拉出来发送到 Scribe 里。

而即使你用了 Kafka，数据都存放在了 Kafka Broker 的本地硬盘上，重放日志的动作你还是少不了的。这个时候，你会面临的问题是，**重放日志需要花费很多时间、或者短时间内**

会消耗很多计算资源。一般来说，你最多也就为流式数据处理，预留平时日志量 3~5 倍的计算能力。那如果你需要重放 30 天的日志，你就需要等上 6~10 天才能重放完。

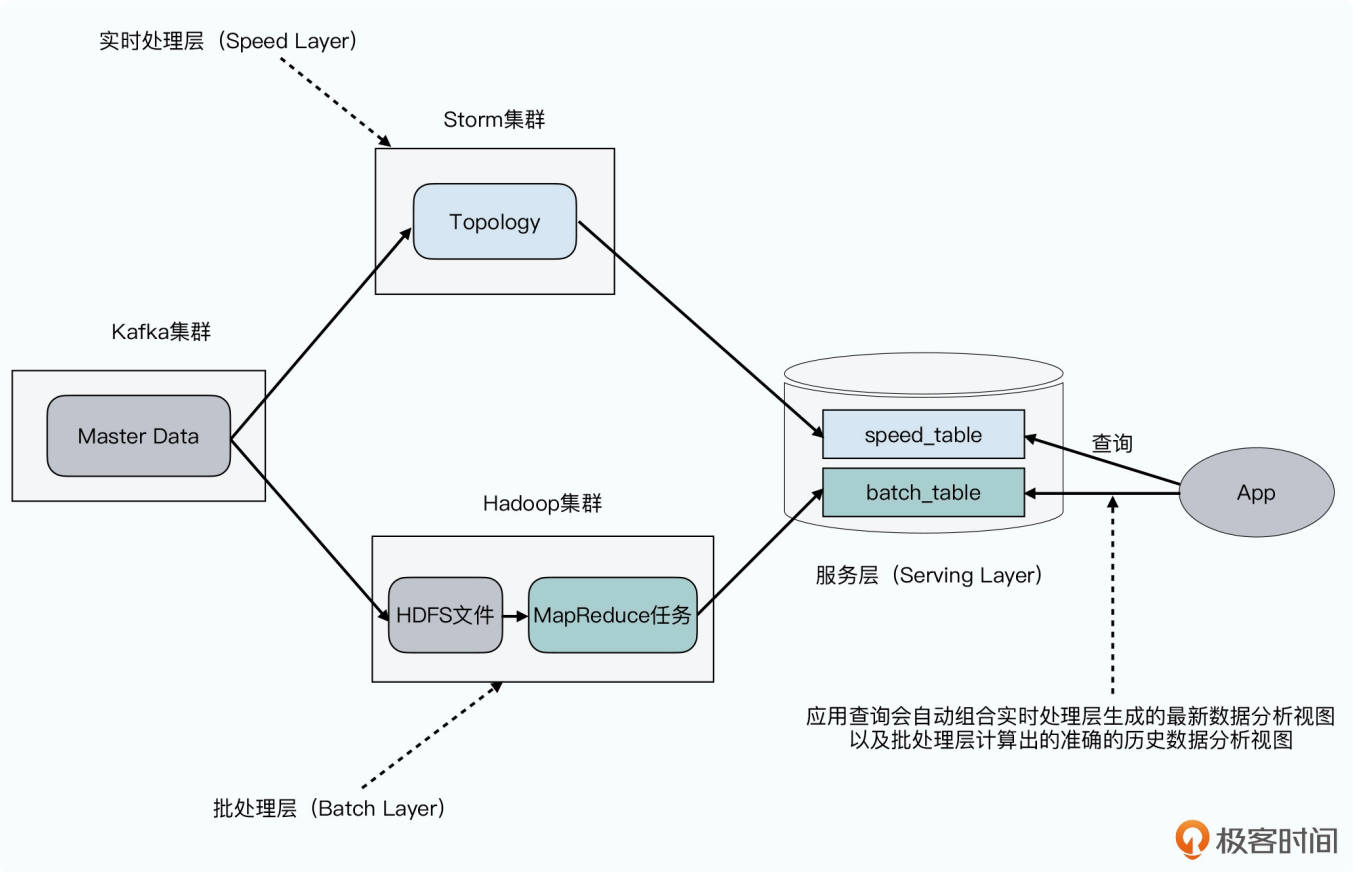
这样就意味着每次修改程序，要么你只能更新新的数据产生的报表，要么你就要等上好几天，才能看到最后的计算结果。

其实，最常会发生的变更，既不来自于硬件故障导致的数据重复处理，也不是来自于业务需求变更导致我们需要修改程序。**最常会发生的变更，来自于解决分析程序里的各种 Bug。**在这种场景下，我们的输入数据不会发生变化，输出的表结构也不会发生变化。但是，我们可能需要反复修改数据处理程序，并且反复在同一份日志数据集上运行这个程序。

这样的程序运行场景，对于大数据的批处理来说，压力并不大，但是对于流式数据处理，一样会有大量重放日志的工作量。

Lambda 架构的基本思想

有鉴于此，Storm 的作者南森·马茨（Nathan Marz）提出了 **Lamda 架构，把大数据的批处理和实时数据结合在一起，变成一个统一的架构。**



Lambda架构

Nathan 的思路是这样的，我们先不去看具体数据是通过什么计算框架来处理的，而是把整个数据处理流程抽象成 **View = Query(Data)** 这样一个函数。我们所有的报表、看板、数据分析结果，都是来自对于原始日志的分析。

所以，原始日志就是我们的主数据（Master Data），不管是 MapReduce 这样的大数据批处理，还是 Storm 这样的大数据流式处理，都是对于原始数据的一次查询（Query）。而这些计算结果，其实就是一个基于特定查询的视图（View）。

当我们的程序有 Bug，其实就是查询写错了，我们的主数据没有变，我们视图的含义也没有变，我们只需要重新写一个查询就好了。而如果我们有需求层面的变更，就是我们需要一个新的视图，以及对应的新的查询了。

而对于实际数据分析系统的用户来说，其实他关心的既不是 Query 也不是 Master Data，而是一个个 **View**。那么，我们在系统的整体架构上，就只需要对这些用户暴露出 View，而不需要告诉他们，具体下面的 Query 和 Master Data 的细节就好了。这样，我们可以按照 Hadoop 和 Storm 本身合适的场景进行选择。

一方面，我们可以**通过 Storm 进行实时的数据处理**，能够尽快获得想要的报表和数据分析结果。另一方面，我们同样会**定时运行 MapReduce 程序**，获得更准确的数据结果。在 MapReduce 程序运行完之前，我们的分析决策基于 Storm 的实时计算结果；但是当 MapReduce 更准确的计算结果出来了，我们就可以拿这个结果替换掉之前的实时计算结果。

而对于外部用户来说，他们看到的始终是同一个视图，只是这个视图，会随着时间的变化不断修正数据结果罢了。

所以，Nathan Marz 总结的 Lambda 结构，是由这样几部分组成的：

第一部分是输入数据，也就是 **Master Data**，这部分也就是我们的原始日志。

然后是一个**批处理层**（Batch Layer）和一个**实时处理层**（Speed Layer），它们分别是一系列 MapReduce 的任务，和一组 Storm 的 Topology，获取相同的输入数据，然后各自计算出自己的计算结果。

最后是一个**服务层**（Serving Layer），通常体现为一个数据库。批处理层的计算结果和实时处理层的结果，会写入到这个数据库里。并且，后生成的批处理层的结果，会不断

替代掉实时处理层的计算结果，也就是对于最终计算的数据进行修正。

对于外部的用户来说，他不需要和批处理层以及实时处理层打交道，而只需要通过像 SQL 这样的查询语言，直接去查询服务层就好了。

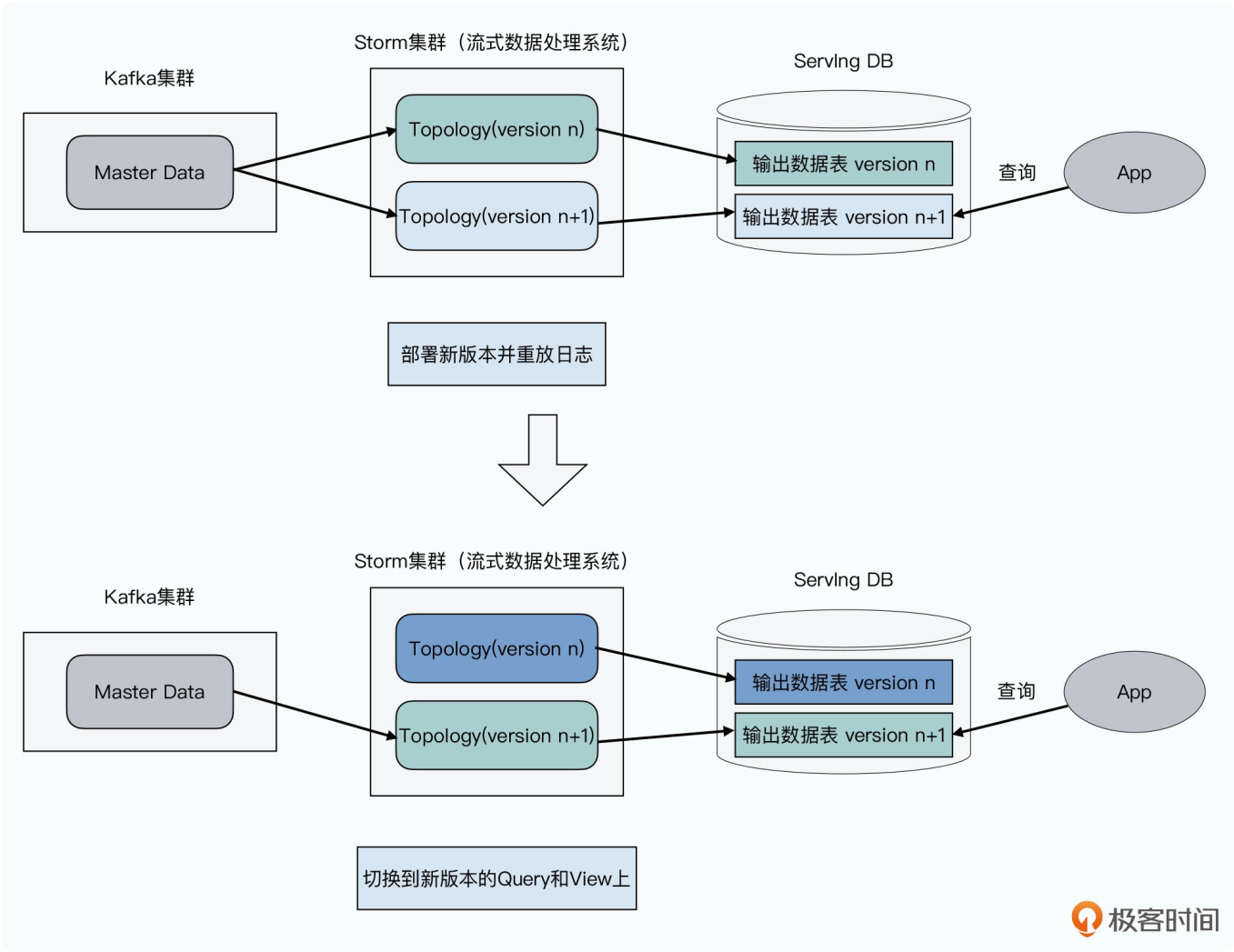
数据处理的 Kappa 架构

可以看到，Lambda 架构很好地结合了 MapReduce 和 Storm 的优点。而这个 Lambda 结构，最终也变成了 Twitter 的一个开源项目 SummingBird。但是，这个 Lambda 架构也有一个显著的**缺点**，也就是什么事情都需要做两遍。

这个做两遍，体现在两个方面：

首先，所有的视图，既需要在实时处理层计算一次，又要在批处理层计算一次。**即使我们没有修改任何程序，也需要双倍的计算资源。**

其次，我们所有的数据处理的程序，也要撰写两遍。MapReduce 的程序和 Storm 的程序虽然要计算的是同样的视图，但是因为底层的框架完全不同，代码我们就需要写两套。**这样意味着，我们需要双倍的开发资源。**



Kappa架构

而且，因为批处理层和实时处理层的代码不同，我们还不得不解决，两遍对于同样视图的理解不同，采用了不同的数据处理逻辑，引入新的 Bug 的问题。

不过，在 Kafka 还没有成熟的时候，把数据分成批处理层和实时处理层是很难避免的。主要问题在于，我们**重放实时处理层的日志是个开销很大的动作**。通过 Scribe 这样的日志收集器，我们的 Master Data 最终是以一个个固定文件落地到 HDFS 的文件系统上。一旦我们想要重放日志，我们就需要把日志从 HDFS 上，分片拉到不同的服务器上，再搭建起多个 Scribe 的集群，去重放日志。

但是，在有了 Kafka 之后，重放日志一下子变得简单了。因为我们所有的日志，都会在 Kafka 集群的本地硬盘上。而通过重放日志来重新进行数据计算，也只是设定一下新的分析程序在 ZooKeeper 上的 Offset 就好了。

有鉴于此，Kafka 的作者杰伊·克雷普斯（Jay Kreps）就提出了一个新的数据计算框架，称之为 **Kappa 架构**。Kappa 架构在 View = Query(Data) 这个基本的抽象理念上，和

Lambda 架构没有变化。但是相比于 Lambda 架构，Kappa 架构去掉了 Lambda 架构的批处理层，而是**在实时处理层，支持了多个视图版本**。

我们之所以要有 View = Query(Data) 这么一个抽象，是因为我们的原始日志，也就是 Data 是不会变化的，而我们想要的 View 也不会变化。但是具体的 Query，可能会因为程序有 Bug 而比较频繁地被修改。

在 Kappa 架构下，如果要对 Query 进行修改，我们原来的实时处理层的代码可以先不用动，而是可以先部署一个新版本的代码，比如一个新的 Topology 上去。然后，我们会对这个 Topology 进行对应日志的重放，在服务层生成一份新的数据结果表，也就是视图的一个新的版本。

在日志重放完成之前，外部的用户仍然会查询旧的实时处理层产生的结果。而一旦日志重放完成，新的 Topology 能够赶上进度，处理到最新产生的日志，那么我们就可以让查询，切换到新的视图版本上来，旧的实时处理层的代码也就可以停掉了。

而随着 Kappa 架构的提出，大数据处理又开始迈入了一个新的阶段，也就是“流批一体”逐步进入主流的阶段。而这个，也是我们接下来的课程中要探讨的主题了。

小结

好了，到这里，我们对于 Kafka、Lambda 架构，以及 Kappa 架构也就学习完了。在今天这节课里，我们看到，Kafka 的分布式架构其实非常简单。

Kafka 本身没有 Master，每一个 Broker 节点都会把自己注册到 ZooKeeper 上。所有的 Broker 本身也不维护任何状态，对应的状态信息也是放在 ZooKeeper 上，而下游的 Consumer 也是一样。对应 Consumer 处理数据到哪里了，就是简单地**在 ZooKeeper 上，为每一个分区维护了一个最新的 Offset**。而对应的数据分区分配给哪一个 Consumer，也是通过 ZooKeeper 里的“**所有权注册表**”记录下来的，分配的逻辑也非常简单，也就是按照分区和 Consumer 的数量，均匀地根据 ID 顺序分配。

而有了 Storm 和 Kafka 之后，工程界开始思考如何将数据的批处理和流式处理统一起来。Storm 的作者 Nathan Marz 提出了一个抽象概念，那就是 View = Query(Data)。我们的数据处理程序，只是针对数据的一个抽象函数，本身没有状态，这个函数运行在一个主数

据集上，可以拿到一个对应的结果视图。所以，基于这个概念，他把自己设计的架构称之为 Lambda 架构。

在 Lambda 架构里，我们的数据处理程序，被分成**批处理层**、**实时处理层**，以及**服务层**。批处理层的结果会不断替换掉实时处理层的计算结果，以不断给出更准确的视图。而外部的应用，只会查询服务层，并不需要关心底层的数据处理的实现是怎么样的。

不过，在 Lambda 架构下，我们的数据需要处理两遍，我们的程序代码也要在不同的计算框架下实现两遍。为了减少这样的双倍开销，Jay Kreps 提出了 Kappa 架构。

Kappa 架构利用了 Kafka 把日志数据保留在 Broker 本地硬盘，重放非常容易这样一个特点，提出了**放弃批处理层，转而在实时处理层提供多个程序版本的思路**。而这个思路，也会是接下来几年里，大数据处理进一步进化的主要方向。

推荐阅读

这节课，我要给你推荐的阅读材料，就是关于 Kappa 架构的。在 Storm 发布之后，很快 Lambda 架构就成为了大数据处理的一个主流的架构设计方案。直到 2014 年，Kafka 的作者 Jay Kreps，撰写了“Questioning the Lambda Architecture”这样一篇文章，讲述了他对于 Lambda 架构不足之处的思考，以及他认为的流式数据处理的 Kappa 架构应该是什么样的。

我推荐你去读一下原文，有助于你理解现代的流式数据处理模式，是怎么样一步步进化过来的。

思考题

在最初的论文里，无论是 Kafka 还是 Storm，都在设计上，只保障了“至少一次”的数据处理模式。那么，我们是否有可能在 Kafka 和 Storm 上，实现“正好一次”的数据处理机制呢？如果要实现这样的机制，整个数据处理流程会是怎么样的？以及它会对 Kafka 和 Storm 带来哪些负面影响呢？

欢迎在留言区分享你的答案和思考，也欢迎你把今天的内容分享给更多的朋友。

上一篇 27 | Kafka（一）：消息队列的新标准

精选留言 (2)

写留言

在路上
2021-12-10

徐老师好，在分布式系统中恰好一次的语义需要两阶段提交的支持，通过协调者记录正在处理哪一条数据，等各个节点确认数据可以被处理之后，在应用处理方案。这个方式最大的问题就是延迟高。如果还需要保持数据处理顺序的话，后面的数据还要排队，吞吐量也受到严重影响。

展开

刘飞

2021-12-10

说的是老版本的kafka吧，新版本kafka消费者不依赖zk