关于Lambda架构的疑问

caolei

Exported on 01/10/2020

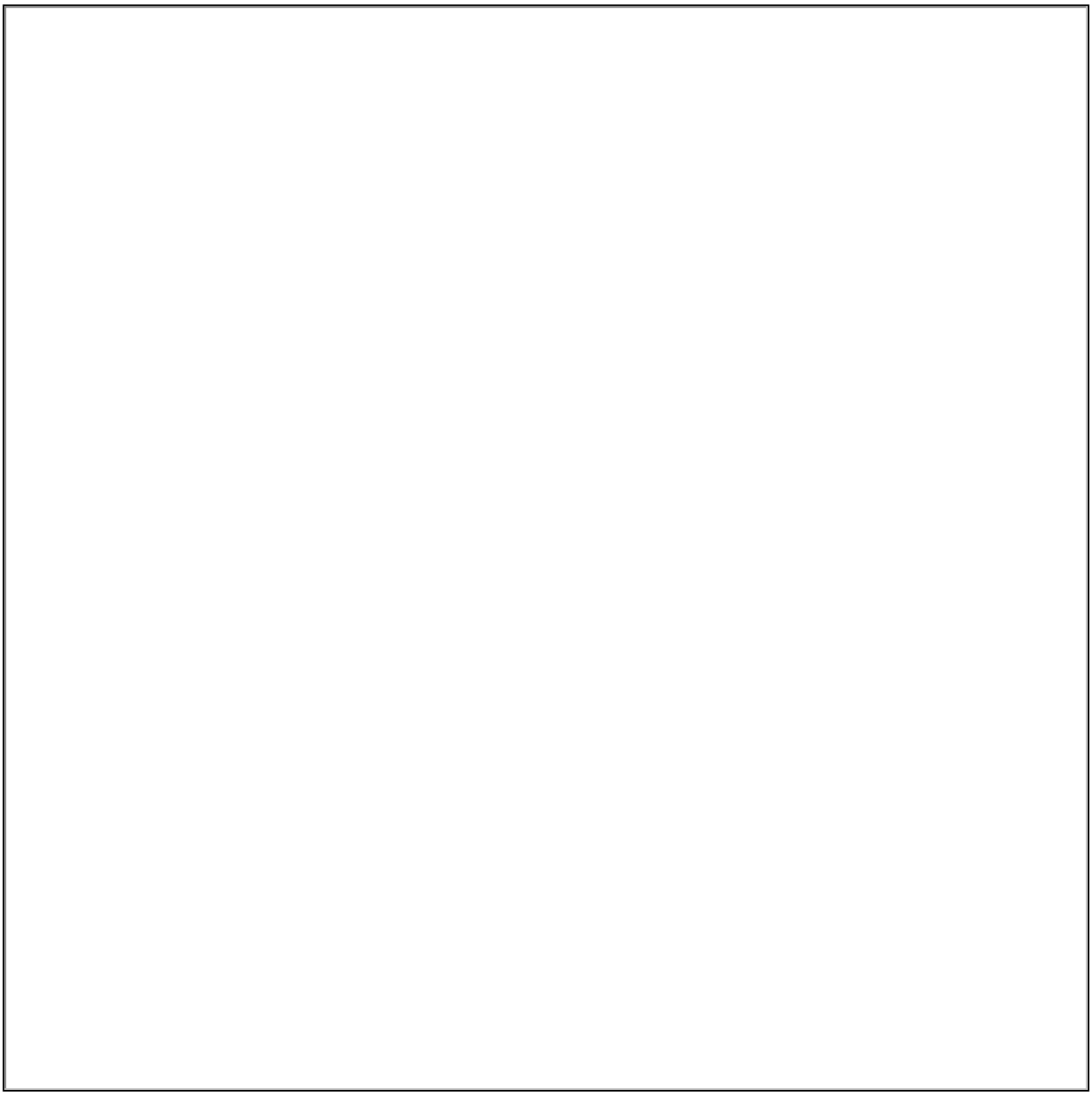
Table of Contents

Nathan Marz[1] 写过一篇很受欢迎的博客文章(“如何击败CAP定理[2]”)，文章中描述了一种被称为Lambda的架构。Lambda架构是在MapReduce和Storm[3]或类似系统之上构建流处理应用程序的一种方法。事实证明，这是一个非常受欢迎的方法，并且有专门的网站[4]和书[5]来传播这种架构方法。由于我一直在使用Kafka[6]和Samza[7]构建LinkedIn的实时数据处理基础设施，所以会被经常问到Lambda架构。我想我应该描述一下我的想法和经历。

1 http://nathanmarz.com/about/
2 http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html
3 http://storm.incubator.apache.org/
4 http://lambda-architecture.net/
5 https://www.manning.com/books/big-data
6 http://kafka.apache.org/
7 http://samza.apache.org/

1 什么是Lambda架构

在正式介绍Lambda架构是什么之前，我们先看下Lambda架构的样子。下图就是是Lambda架构的通用模型。



由上图我们可知：Lambda架构中会将数据(不可变记录)并行的灌入到批处理系统和流处理系统。分别在这两个系统进行数据转换处理(一次在批处理系统中，一次在流处理系统中)。然后将处理结果分别送到一个被称为Serving DB的地方，用户在查询时可以将两个系统的结果拼接在一起，用以生成最终的查询结果。

关于Lambda架构其实有很多变种，在这里我有意进行了简化。例如上图中的Kafka、Storm、Hadoop,您都可以换成其他功能类似的组件。比如Spark、Flink、Tez等。另外对于Serving DB，人们也通常使用两个不同的数据库来存储输出表，一个为实时优化，另一个为批量更新优化。

Lambda架构的目标是围绕复杂的异步转换构建的应用程序，这些应用系统需要以低延迟运行(例如，几秒钟到几小时)。一个很好的例子是新闻推荐系统，它需要抓取各种新闻源，处理和规范化所有输入，然后索引、排序，并将其存储起来便于提供服务。

我在LinkedIn参与了许多实时数据系统和管道的建设。其中一些方法就是以这种方式工作的，仔细想想，这并不是我最喜欢的方法。我认为有必要描述下我所看到的这种体系结构的优缺点，并给出我更喜欢的替代方案。

2 Lambda架构的优势

Lambda架构强调保持输入数据的不可变性，这点我非常喜欢并赞同。我认为将数据转换建模为从原始输入到一系列物化阶段的有很多优点。这是使大型MapReduce工作流易于处理的原因之一，因为它使您能够独立调试每个阶段。我认为这一点也很好地被用于流处理领域。关于捕获和转换不可变数据流的想法，我之前写过一篇文章[8]对其进行了介绍。

我还喜欢这种体系结构突出的数据再处理问题。再处理是流处理的关键挑战之一，但常常被忽略。所谓“再处理”，我的意思是重新处理输入数据以重新获得输出。这是一个非常明显但经常被忽略的要求。代码总是会改变的。因此，如果您有从输入流派生输出数据的代码，每当代码发生更改时，您都需要重新计算输出以查看代码更改的效果。

为什么代码会发生改变？因为您的应用程序在发展，您希望计算以前不需要的新输出字段。或者因为您发现了一个bug，需要修复它。无论如何，当它发生变化时，您需要重新生成输出。我发现，许多试图构建实时数据处理系统的人并没有在这个问题上花太多心思，最终得到的只是一个无法快速发展的系统，因为它没有方便的方法来处理数据再处理问题。Lambda体系结构强调了这个问题，这一点值得赞扬。

针对Lambda架构的出现，其实还有许多其他的动机，但是我认为它们没有多大意义。其中一个说是实时处理本质上是近似的，比批处理功能更弱，更容易丢失数据。我不认为这是真的。但不可否认，现有的一些流处理框架不如MapReduce成熟，但是，流处理系统没有理由不能提供像批处理系统那样强大的语义保证。

我听到的另一种解释是Lambda架构允许不同的数据系统使用不同的trade-offs，从而在某种程度上“击败了CAP定理”。长话短说，虽然流处理中确实存在延迟和可用性之间的trade-offs，但这是异步处理体系结构，因此计算的结果不会立即与传入的数据保持一致。遗憾的是，CAP定理仍然完好无损。

8 http://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying

3 Lambda架构的劣势

Lambda体系结构的问题在于需要维护在两个复杂的分布式系统中生成相同结果的代码，看起来就很痛苦。并且我认为这个问题是无法解决的。

在Storm和Hadoop这样的分布式框架中编程是很复杂的。不可避免地，代码最终会被设计成和它所运行的框架强相关。实现Lambda体系结构系统的操作复杂性似乎是每个人都一致同意的。

为什么不能改进流处理系统来处理目标域中的全部问题集?解决此问题的一种建议方法是使用一种语言或框架，该语言或框架对实时和批处理框架进行抽象。您使用这个高级框架编写代码，然后它“向下编译”以在幕后进行流处理或MapReduce。Summingbird[9]就是这样一个框架。这确实让事情好了一点，但我不认为它解决了问题。

最后，即使可以避免对应用程序进行两次编码，运行和调试两个系统的操作负担也会非常高。任何新的抽象都只能提供这两个系统交集所支持的特性。更糟的是，致力于这种新的超级框架将使Hadoop如此强大的工具和语言的丰富生态系统隔离开来(Hive、Pig、Crunch、Cascading、Oozie等)。

打个比方，考虑一下让跨数据库ORM真正透明所面临的众所周知的困难。考虑到这只是对非常相似的系统进行抽象的问题，这些系统使用(几乎)标准化的接口语言提供几乎相同的功能。这在几乎不稳定的分布式系统上构建完全不同的编程范例上进行抽象的问题要困难得多。

9 http://github.com/twitter/summingbird

4 我们做过的

实际上，在LinkedIn已经经历了好几轮这种情况。我们已经构建了各种混合的Hadoop体系结构，甚至还构建了一个特定领域的API，它允许代码“透明地”运行在实时或Hadoop中。这些方法都奏效了，但没有一种非常令人愉快或富有成效。保持两个不同系统中编写的代码完全同步是非常非常困难的。用于隐藏底层框架的API被证明是最容易泄漏的抽象。它最终需要深入了解Hadoop知识以及实时层的知识——并添加了新的需求，即当您调试问题或试图解释性能问题时，您必须充分了解API如何转换到这些底层系统。最终需要深入的Hadoop知识以及实时层的知识，如果需要添加一个新的功能，即无论何时调试问题或试图推断性能时，您都要充分了解如何将API转换为这些底层系统。

现在，我的建议是，如果您对延迟不敏感，那么使用批处理框架(如MapReduce)，如果您对延迟敏感，那么使用流处理框架，但是除非绝对必要，否则不要尝试同时使用这两种处理框架。

那么，为什么Lambda架构如此令人兴奋呢? 我认为原因是人们越来越需要构建复杂的、低延迟的处理系统。他们手头上的两种工具不能完全解决问题：一个可伸缩的高延迟批处理系统可以处理历史数据，另一个低延迟流处理系统不能重新处理结果。通过管道胶带把这两种东西粘在一起，他们实际上可以建立一个可行的解决方案。

从这个意义上说，尽管它可能很痛苦，但我认为Lambda体系结构解决了一个通常被忽略的重要问题。但我不认为这是一个新的范式或大数据的未来。它只是由现有工具的当前限制所驱动的临时状态。我也认为还有更好的选择。

5 另一种选择

作为一个设计基础设施的人，我认为最突出的问题是:为什么不能改进流处理系统来处理其目标领域中的全部问题集?为什么需要粘在另一个系统上? 为什么不能同时进行实时处理和代码更改时的再处理? 流处理系统已经有了并行的概念; 为什么不通过增加并行度和非常快地重放历史来处理重新处理呢? 答案是可以这么做。我认为这实际上是一个合理的替代架构，如果你现在正在构建这种类型的系统。

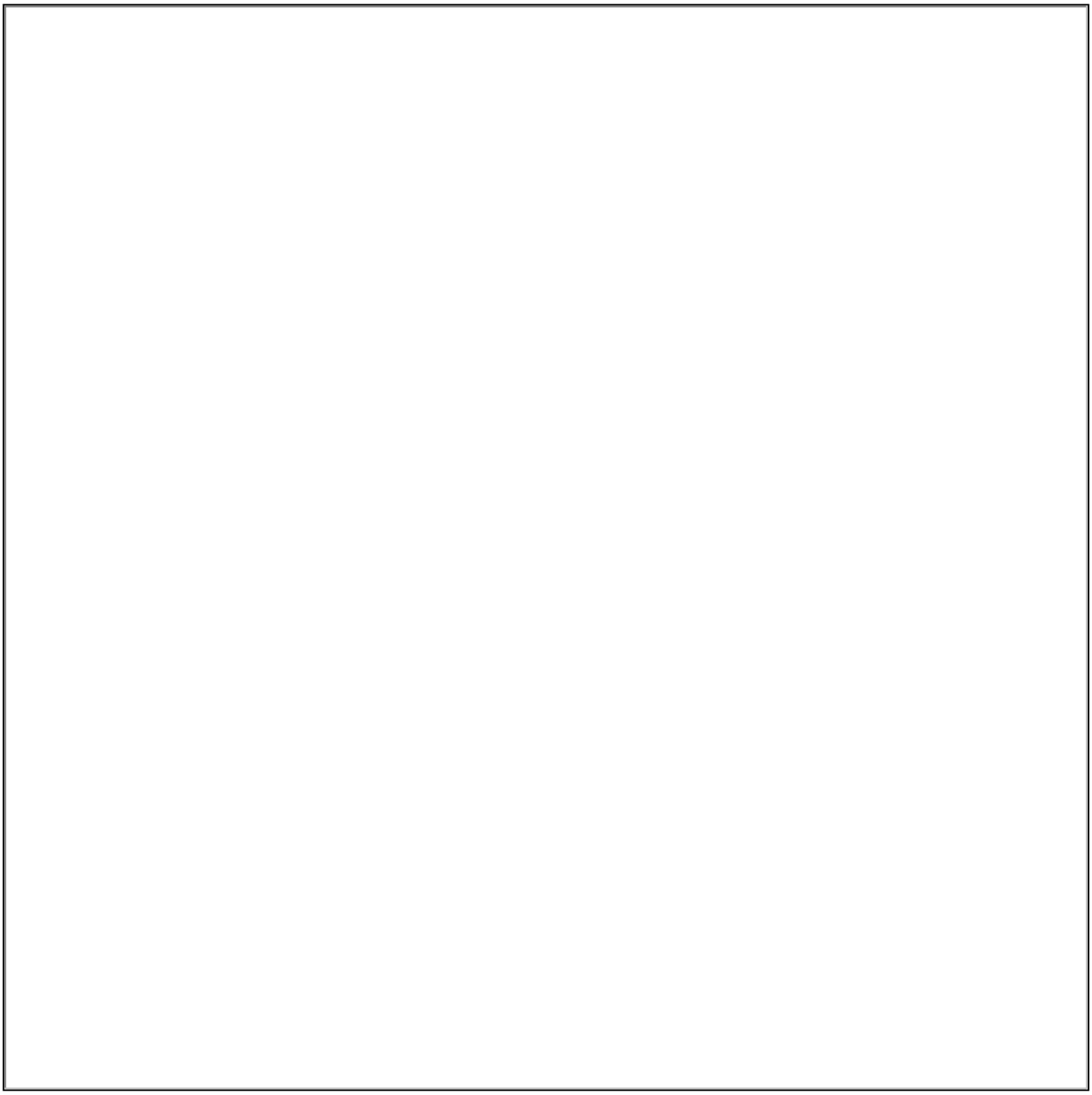
当我与人们讨论这个问题时，他们有时会告诉我，流处理不适用于历史数据的高吞吐量处理。但我认为这是一种基于他们所使用系统局限性的直觉，主要是，这些系统要么伸缩性很差，要么无法保存历史数据。这给他们一种感觉，流处理系统本质上是计算一些临时流的结果，然后丢弃所有底层数据的东西。但没有理由相信这是应该的。流处理中的基本抽象是数据流DAGs，它与传统数据仓库(Volcano[10])中的底层抽象完全相同，也是MapReduce后续Tez[11]中的基本抽象。流处理只是这个数据流模型的泛化，它向最终用户公开中间结果的检查点和连续输出。

那么，我们如何直接在流处理系统中进行再处理操作呢?我喜欢的方法其实非常简单:

1. 使用Kafka或其他系统，可以保留希望能够重新处理的数据的完整日志，并允许多个订阅者。例如，如果您希望重新处理最多30天的数据，请将Kafka中的保留时间设置为30天。
2. 当您希望进行再处理时，启动流处理作业的第二个实例，该实例从保留数据的开始处开始处理，但将此输出数据定向到一个新的输出表。
3. 当第二个作业完成时，将应用程序切换为从新表读取。
4. 停止作业的旧版本，并删除旧的输出表。

下图简单展示了上面的处理逻辑:

10 http://paperhub.s3.amazonaws.com/dace52a42c07f7f8348b08dc2b186061.pdf
11 http://hortonworks.com/hadoop/tez/



与Lambda体系结构不同，在这种方法中，您只需要在处理代码更改时进行重新处理，并且实际上需要重新计算结果。当然，重新计算的工作只是相同代码的改进版本，运行在相同的框架上，获取相同的输入数据。当然，您会希望提高再处理作业的并行性，以便它能够非常快地完成。

也许我们可以称它为Kappa架构，尽管这个想法可能太简单了，不值得用希腊字母来写。

当然，您可以进一步优化它。在许多情况下，您可以组合这两个输出表。然而，我认为在短时间内同时拥有两者是有好处的。这允许您通过一个将应用程序重定向到旧表的按钮立即恢复到旧逻辑。在特别重要的情况下(比如，您的广告目标标准)，您可以使用自动A/B测试或bandit算法[12]来控制切换，以确保与以前的版本相比，您正在推出的任何bug修复或代码改进都没有意外地降低性能。

12 http://shop.oreilly.com/product/0636920027393.do

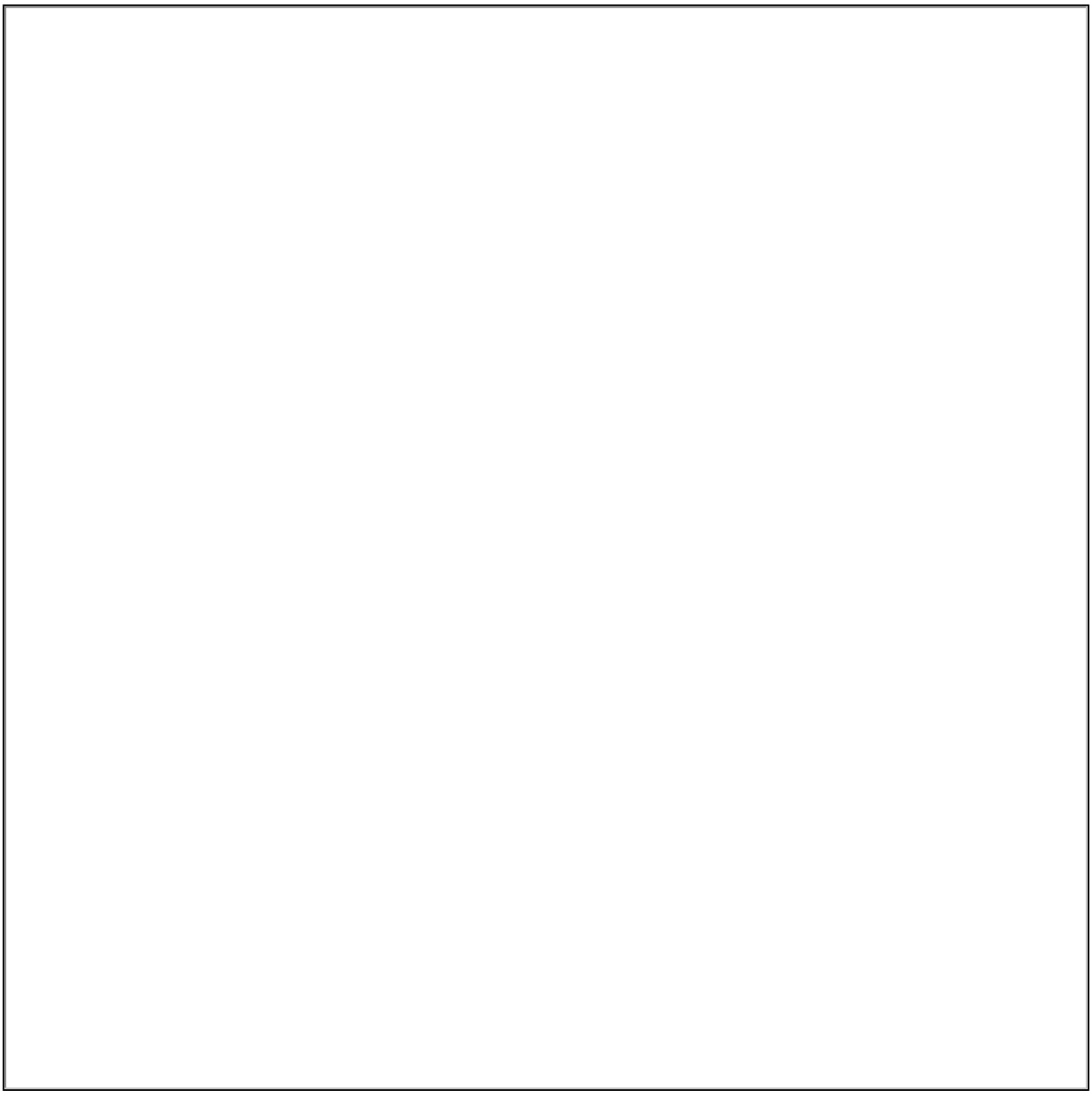
注意，这并不意味着您的数据不能进入HDFS;这就意味着你不会在那里进行再处理。Kafka与Hadoop有很好的集成，因此将任何Kafka主题镜像到HDFS中都很容易。在Hadoop中，流处理作业的输出流，甚至中间流，用于Hive之类的工具中的分析，或者用作其他离线数据处理流的输入，这通常很有用。

我们在这篇文章中[13]已经记录了使用Samza实现此方法以及对再处理体系结构的其他变体。

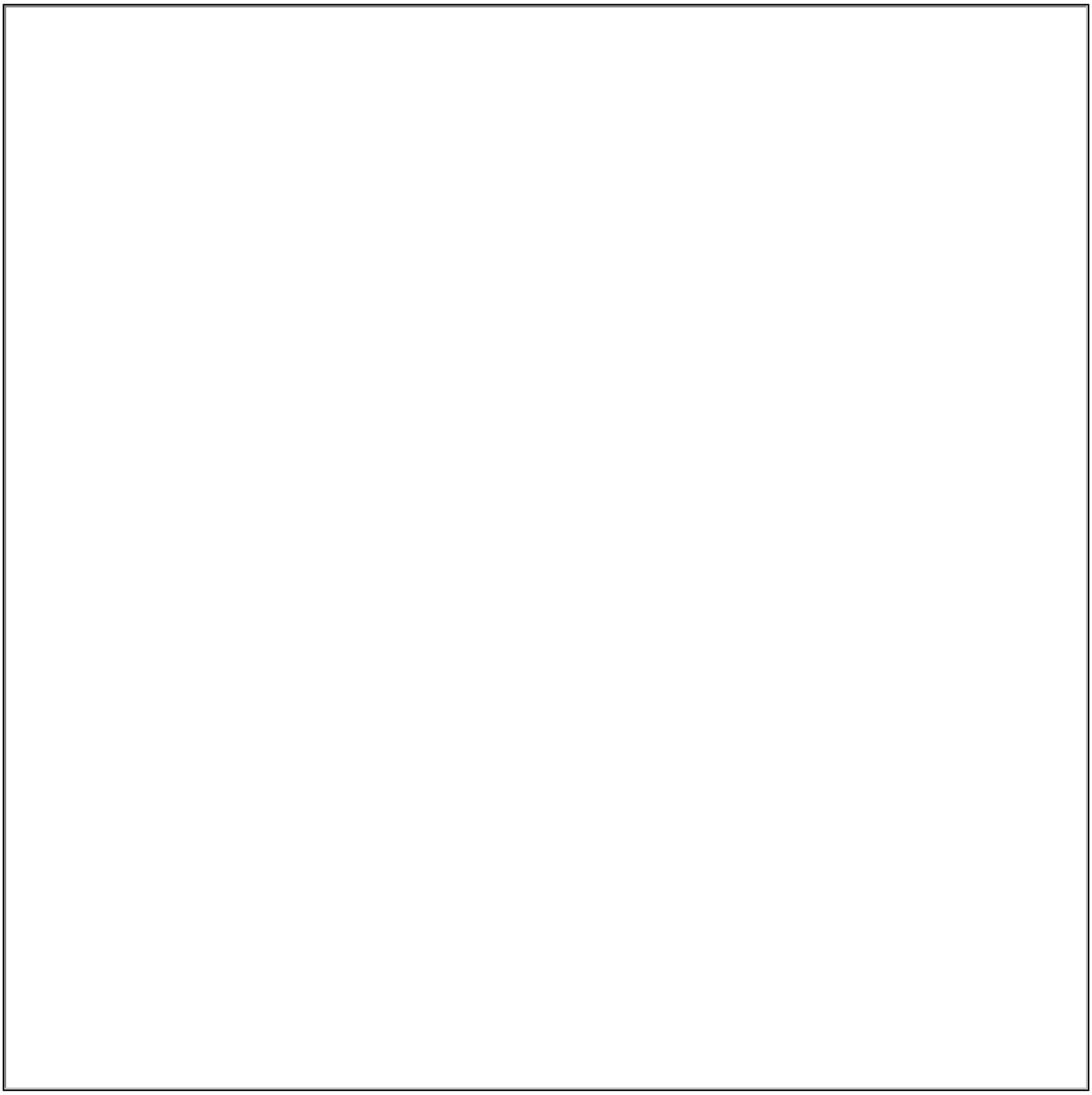
13 http://samza.apache.org/learn/documentation/0.7.0/jobs/reprocessing.html

6 一些背景知识

对于那些不太熟悉Kafka的人来说，我刚才所描述的可能没有意义。快速复习一下就有希望把事情弄清楚。
Kafka是这样维护有序日志的:



Kafka中的“主题”其实就是这些日志的集合:



使用此数据的流处理器只维护一个“偏移量”，即它在每个分区上处理的最后一条记录的日志条目号。因此，更改使用者返回并重新处理数据的位置与使用不同的偏移量重新启动作业一样简单。为相同的数据添加第二个使用者只是指向日志中不同位置的另一个读取器。

Kafka支持复制和容错，运行在廉价的普通硬件上，并且乐于为每台机器存储许多TB级别的数据。因此，保留大量数据是一件非常自然和经济的事情，不会影响性能。LinkedIn在网上保存了超过PB的Kafka存储，许多应用程序都很好地利用了这种长时间的保留模式。

廉价的消费者和保留大量数据的能力使得添加第二个“再处理”任务只需要启动代码的第二个实例，但是要从日志中的不同位置开始。

这种设计并非偶然。我们构建Kafka的目的是使用它作为流处理的根基，我们想要的正是这个处理数据再处理的模型。好奇的读者可以在这里找到更多关于Kafka的信息[14]。

然而，从根本上说，没有任何东西把这个想法与Kafka联系在一起。您可以替换任何支持长时间保留有序数据的系统(例如HDFS或某种数据库)。实际上，很多人都熟悉类似的模式，即事件源或CQRS。当然，分布式数据库的人会告诉你，这只是对物化视图维护的一个轻微的重新命名，他们会很高兴地提醒你，他们很久以前就知道了。

14 https://kafka.apache.org/documentation.html#introduction

7 对照

我知道使用Samza作为流处理系统可以很好地使用这种方法，因为我们在LinkedIn上就是这么做的。但是我不知道它在Storm或其他流处理系统中不能正常工作的原因。我对Storm还不是很熟悉，所以我很高兴听到其他人已经在这么做了。在任何情况下，我认为总体思想和系统是相对独立的。

这两种方法之间的效率和资源权衡多少有些勉强。Lambda架构需要一直运行再处理和实时处理，而我所建议的只需要在需要重新处理时运行作业的第二个副本。然而，我的建议要求在输出数据库中临时拥有2倍的存储空间，并要求数据库支持用于重新加载的大容量写操作。在这两种情况下，额外的再处理负荷可能会平均出来。如果您有许多这样的作业，它们就不会同时进行再处理，因此在一个包含几十个这样的作业的共享集群中，您可能会为在任何给定时间内正在积极进行再处理的少数作业额外预算几个百分点的容量。

真正的优势根本不在于效率，而在于允许人们在一个处理框架上开发、测试、调试和操作他们的系统。因此，在简单性很重要的场景下，可以将此方法视为Lambda体系结构的替代方案。

原文地址：https://www.oreilly.com/ideas/questioning-the-lambda-architecture