

大数据凉了？No，流式计算浪潮才刚刚开始！

原创：AI前线小组 译 AI前线 2 days ago



big data

策划编辑 | Natalie

翻译 | 巴真

编辑 | Debra

AI 前线导读：本文重点讨论了大数据系统发展的历史轨迹，行文轻松活泼，内容通俗易懂，是一篇茶余饭后用来作为大数据谈资的不严肃说明文。本文翻译自《Streaming System》最后一章《The Evolution of Large-Scale Data Processing》，在探讨流式系统方面本书是市面上难得一见的深度书籍，非常值得学习。

更多干货内容请关注微信公众号“AI 前线”（ID：ai-front）

大规模数据处理的演化历程

大数据如果从 Google 对外发布 MapReduce 论文算起，已经前后跨越十五年，我打算在本文和你蜻蜓点水般一起浏览下大数据的发展史，我们从最开始 MapReduce 计算模型开始，一路走马观花看看大数据这十五年关键发展变化，同时也顺便会讲解流式处理这个领域是如何发展到今天的这幅模样。这其中我也会加入一些我对一些业界知名大数据处理系统（可能里面有些也不那么出名）的观察和评论，同时考虑到我很有可能简化、低估甚至于忽略了很多重要的大数据处理系统，我也会附带一些参考材料帮助大家学习更多更详细的知识。

另外，我们仅仅讨论了大数据处理中偏 MapReduce/Hadoop 系统及其派系分支的大数据处理。我没有讨论任何 SQL 引擎 [1]，我们同样也没有讨论 HPC 或者超级计算机。尽管我这章的标题听上去领域覆盖非常广泛，但实际上我仅仅会讨论一个相对比较垂直的大数据领域。

同样需要提醒的一件事情是，我在本文里面或多或少会提到一些 Google 的技术，不用说这是因为与我在谷歌工作了十多年的经历有关。但还有另外两个原因：1) 大数据对谷歌来说一直很重要，因此在那里创造了许多有价值的东西值得详细讨论，2) 我的经验一直是谷歌以外的人似乎更喜欢学习 Google 所做的事情，因为 Google 公司在这方面一直有点守口如瓶。所以，当我过分关注我们一直在“闭门造车”的东西时，姑且容忍下我吧。

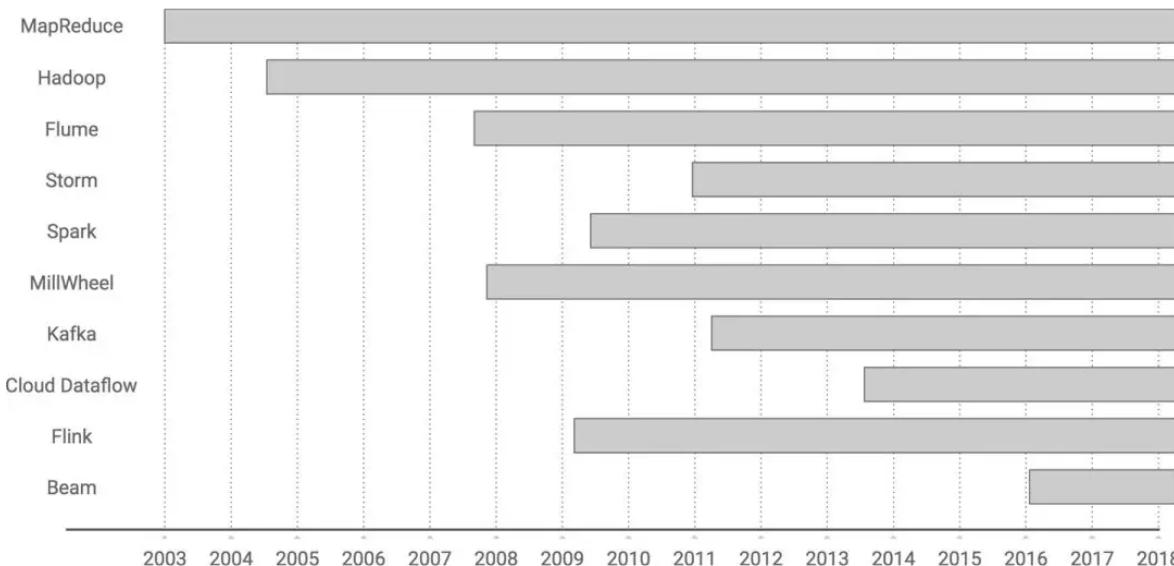


图 10-1 本章讨论各个大数据系统时间表

为了使我们这一次大数据旅行显得更加具体有条理，我们设计了图 10-1 的时间表，这张时间表概括地展示了不同系统的诞生日期。

在每一个系统介绍过程中，我会尽可能说明清楚该系统的简要历史，并且我会尝试从流式处理系统的演化角度来阐释该系统对演化过程的贡献。最后，我们将回顾以上系统所有的贡献，从而全面了解上述系统如何演化并构建出现代流式处理系统的。

MapReduce

我们从 MapReduce 开始我们的旅程。

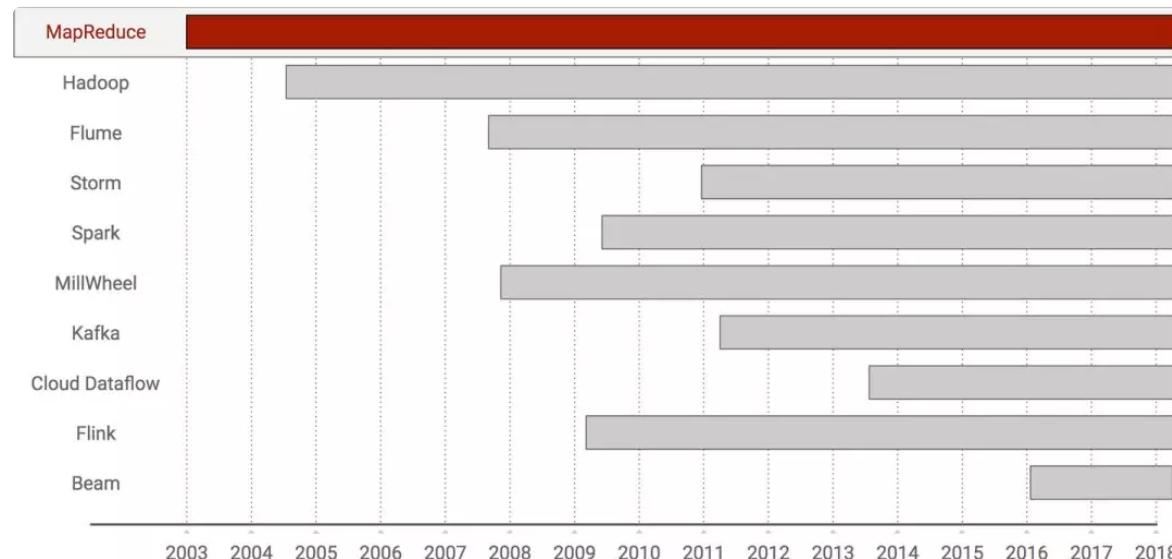


图 10-2 MapReduce 的时间表

我认为我们可以很确定地说，今天我们讨论的大规模数据处理系统都源自于 2003 年 MapReduce。当时，谷歌的工程师正在构建各种定制化系统，以解决互联网时代下大数据处理难题。当他们这样尝试去解决这些问题时候，发现有三个难以逾越的坎儿：

- 数据处理很难 只要是数据科学家或者工程师都很清楚。如果你能够精通于从原始数据挖掘出对企业有价值的信息，那这个技能能够保你这辈子吃喝不愁。
- 可伸缩性很难 本来数据处理已经够难了，要从大规模数据集中挖掘出有价值的数据更加困难。
- 容错很难 要从大规模数据集挖掘数据已经很难了，如果还要想办法在一批廉价机器构建的分布式集群上可容错地、准确地方式挖掘数据价值，那真是难于上青天了。

在多种应用场景中都尝试解决了上述三个问题之后，Google 的工程师们开始注意到各自构建的定制化系统之间颇有相似之处。最终，Google 工程师悟出来一个道理：如果他们能够构建一个可以解决上述问题二和问题三的框架，那么工程师就将可以完全放下问题二和三，从而集中精力解决每个业务都需要解决的问题一。于是，MapReduce 框架诞生了。

MapReduce 的基本思想是提供一套非常简洁的数据处理 API，这套 API 来自于函数式编程领域的两个非常易于理解的操作：map 和 reduce（图 10-3）。使用该 API 构建的底层数据流将在这套分布式系统框架上执行，框架负责处理所有繁琐的可扩展性和容错性问题。可扩展性和容错性问题对于分布式底层工程师来说无疑是非常有挑战的课题，但对于普通工程师而言，无益于是灾难。

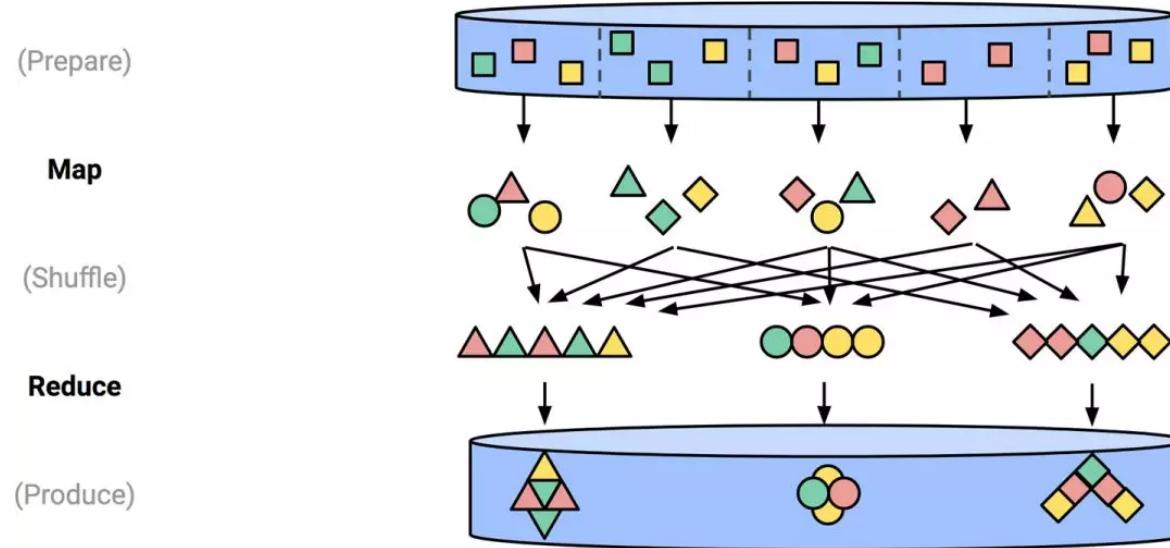


图 10-3 MapReduce 作业原理图

我们已经在第 6 章详细讨论了 MapReduce 的语义，所以我们在此不再赘述。仅仅简单地回想一下，我们将处理过程分解为六个离散阶段（MapRead，Map，MapWrite，ReduceRead，Reduce，ReduceWrite）作为对于流或者表进行分析的几个步骤。我们可以看到，整体上 Map 和 Reduce 阶段之间差异其实也不大；更高层次来看，他们都做了以下事情：

- 从表中读取数据，并转换为数据流（译者注：即 MapRead、ReduceRead）
- 针对上述数据流，将用户编写业务处理代码应用于上述数据流，转换并形成新的一个数据流。（译者注：即 Map、Reduce）
- 将上述转换后的流根据某些规则分组，并写出到表中。（译者注：即 MapWrite、ReduceWrite）

随后，Google 内部将 MapReduce 投入生产使用并得到了非常广泛的业务应用，Google 认为应该和公司外的同行分享我们的研究成果，最终我们将 MapReduce 论文发表于 OSDI 2004（见图 10-4）。

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* op-

图 10-4 MapReduce 论文发表在 OSDI 2004 上

论文中，Google 详细描述了 MapReduce 项目的历史，API 的设计和实现，以及有关使用了 MapReduce 框架的许多不同生产案例的详细信息。当然，Google 没有提供任何实际的源代码，以至于

最终 Google 以外的人都认为：“是的，这套系统确实牛啊！”，然后立马回头去模仿 MapReduce 去构建他们的定制化系统。

在随后这十年的过程中，MapReduce 继续在谷歌内部进行大量开发，投入大量时间将这套系统规模推进到前所未有的水平。如果读者朋友希望了解一些更加深入更加详细的 MapReduce 说明，我推荐由我们的 MapReduce 团队中负责扩展性、性能优化的大牛 Marián Dvorský 撰写的文章《History of massive-scale sorting experiments at Google》（图 10-5）



History of massive-scale sorting experiments at Google

Thursday, February 18, 2016

We've tested MapReduce by sorting large amounts of random data ever since we created the tool. We like sorting, because it's easy to generate an arbitrary amount of data, and it's easy to validate that the output is correct.

Even the [original MapReduce paper](#) reports a TeraSort result. Engineers run 1TB or 10TB sorts as regression tests on a regular basis, because obscure bugs tend to be more visible on a large scale. However, the real fun begins when we increase the scale even further. In this post I'll talk about our experience with some petabyte-scale sorting experiments we did a few years ago, including what we believe to be the largest MapReduce job ever: a 50PB sort.

These days, GraySort is the large scale sorting benchmark of choice. In GraySort, you must sort at least 100TB of data (as 100-byte records with the first 10 bytes being the key), lexicographically, as fast as possible. The site [sortbenchmark.org](#) tracks official winners for this benchmark. We never entered the official competition.

MapReduce happens to be a good fit for solving this problem, because the way it implements reduce is by sorting the keys. With the appropriate (lexicographic) sharding function, the output of MapReduce is a sequence of files comprising the final sorted dataset.

Once in awhile, when a new cluster in a datacenter came up (typically for use by the search indexing team), we in the MapReduce team got the opportunity to play for a few weeks before the real workload moved in. This is when we had a chance to "burn in" the cluster, stretch the limits of the hardware, destroy some hard drives, play with some really expensive equipment, learn a lot about system performance, and, win (unofficially) the sorting benchmark.

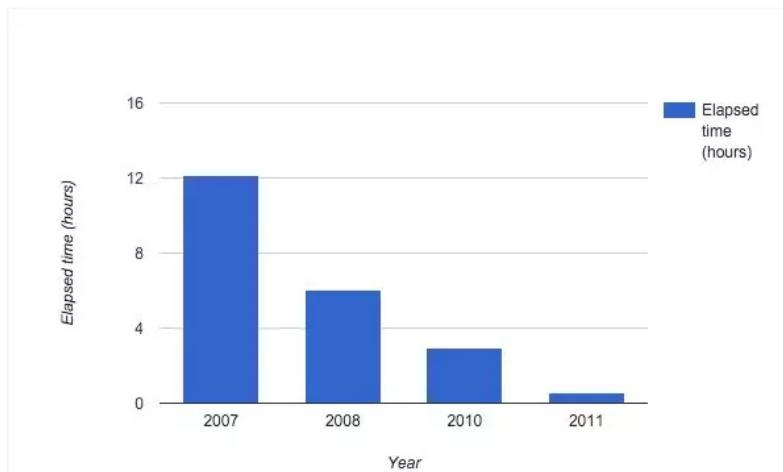


图 10-5 MariánDvorský的《History of massive-scale sorting experiments》博客文章

我这里希望强调的是，这么多年来看，其他任何的分布式架构最终都没有达到 MapReduce 的集群规模，甚至在 Google 内部也没有。从 MapReduce 诞生起到现在已经跨越十载之久，都未能看到真正能够超越 MapReduce 系统规模的另外一套系统，足见 MapReduce 系统之成功。14 年的光阴看似不长，对于互联网行业已然永久。

从流式处理系统来看，我想为读者朋友强调的是 MapReduce 的简单性和可扩展性。MapReduce 给我们的启发是：MapReduce 系统的设计非常勇于创新，它提供一套简便且直接的 API，用于构建业务复杂但可靠健壮的底层分布式数据 Pipeline，并足够将这套分布式数据 Pipeline 运行在廉价普通的商用服务器集群之上。

Hadoop

我们大数据旅程的下一站是 Hadoop（图 10-6）。需要着重说明的是：我为了保证我们讨论的重心不至于偏离太多，而压缩简化讨论 Hadoop 的内容。但必须承认的是，Hadoop 对我们的行业甚至整个世界的影响不容小觑，它带来的影响远远超出了我在此书讨论的范围。

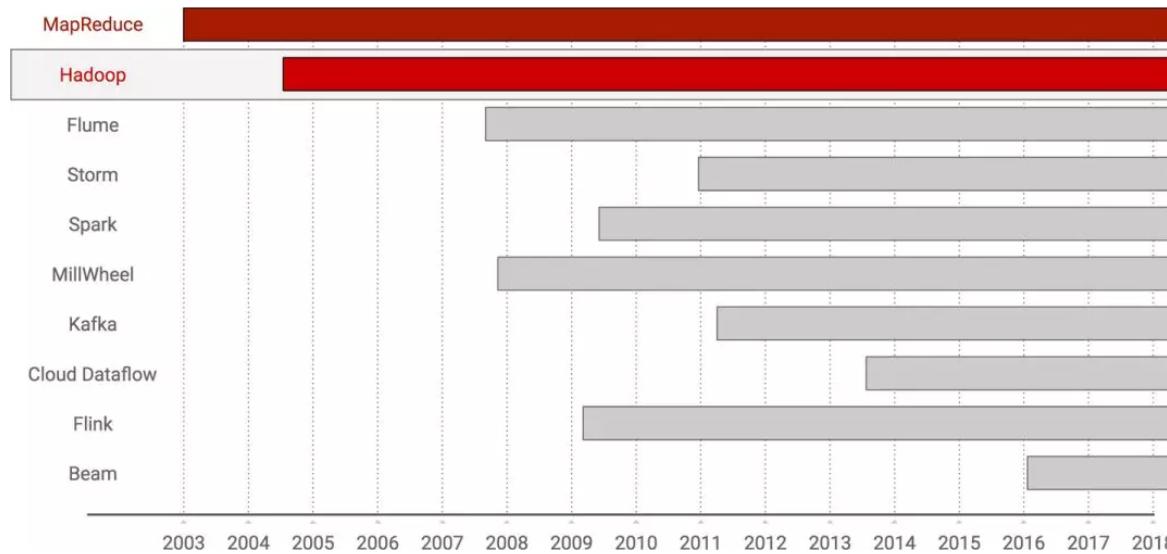


图 10-6 Hadoop 的时间表

Hadoop 于 2005 年问世，当时 Doug Cutting 和 Mike Cafarella 认为 MapReduce 论文中的想法太棒了，他们在构建 Nutch webcrawler 的分布式版本正好需要这套分布式理论基础。在这之前，他们已经实现了自己版本的 Google 分布式文件系统（最初称为 Nutch 分布式文件系统的 NDFS，后来改名为 HDFS 或 Hadoop 分布式文件系统）。因此下一步，自然而然的，基于 HDFS 之上添加 MapReduce 计算层。他们称 MapReduce 这一层为 Hadoop。

Hadoop 和 MapReduce 之间的主要区别在于 Cutting 和 Cafarella 通过开源（以及 HDFS 的源代码）确保 Hadoop 的源代码与世界各地可以共享，最终成为 Apache Hadoop 项目的一部分。雅虎聘请 Cutting 来帮助将雅虎网络爬虫项目升级为全部基于 Hadoop 架构，这个项目使得 Hadoop 有效提升了生产可用性以及工程效率。自那以后，整个开源生态的大数据处理工具生态系统得到了蓬勃发展。与

MapReduce 一样，相信其他人已经能够比我更好地讲述了 Hadoop 的历史。我推荐一个特别好的讲解是 Marko Bonaci 的《The history of Hadoop》，它本身也是一本已经出版的纸质书籍（图 10-7）。

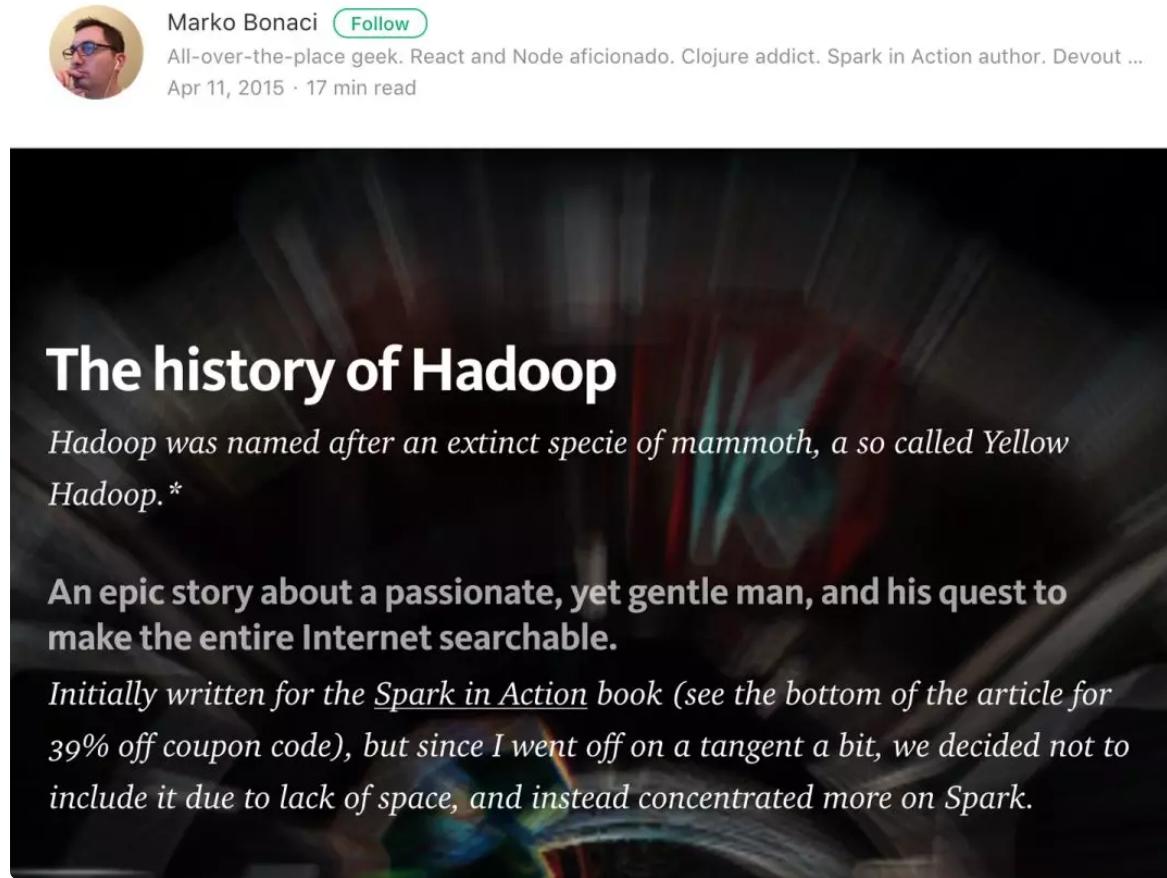


图 10-7 Marko Bonaci 的《The history of Hadoop》

在 Hadoop 这部分，我期望读者朋友能够了解到围绕 Hadoop 的开源生态系统对整个行业产生的巨大影响。通过创建一个开放的社区，工程师可以从早期的 GFS 和 MapReduce 论文中改进和扩展这些想法，这直接促进生态系统的蓬勃发展，并基于此之上产生了许多有用的工具，如 Pig，Hive，HBase，Crunch 等等。这种开放性是导致我们整个行业现有思想多样性的关键，同时 Hadoop 开放性生态亦是直接促进流计算系统发展。

Flume

我们现在再回到 Google，讨论 Google 公司中 MapReduce 的官方继承者：Flume（[图 10-8]，有时也称为 FlumeJava，这个名字起源于最初 Flume 的 Java 版本。需要注意的是，这里的 Flume 不要与 Apache Flume 混淆，这部分是面向不同领域的东西，只是恰好有同样的名字）。

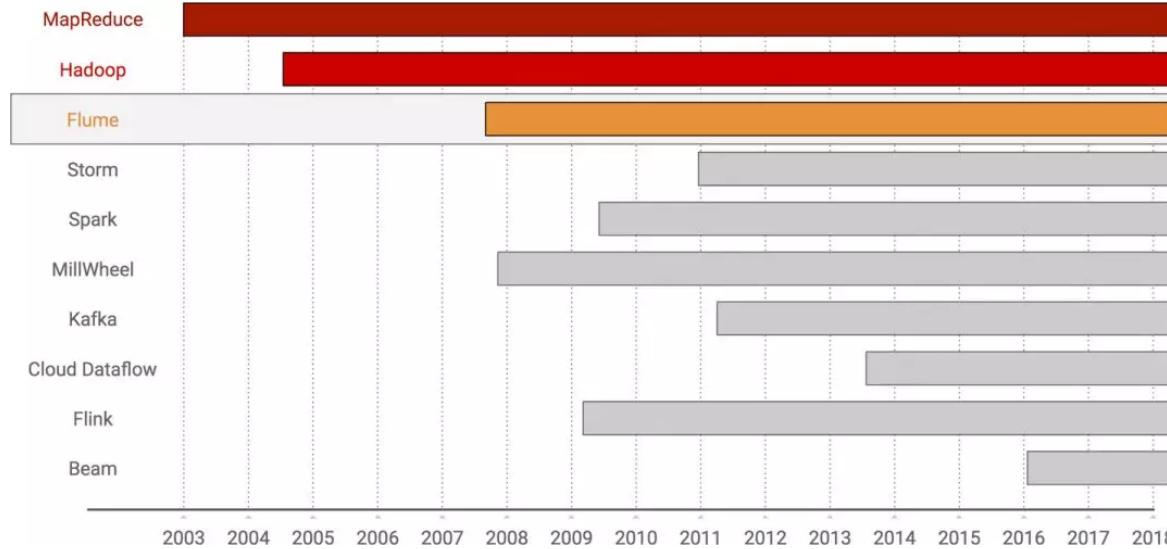


图 10-8 Flume 的时间表

Flume 项目由 Craig Chambers 在 2007 年谷歌西雅图办事处成立时发起。Flume 最初打算是希望解决 MapReduce 的一些固有缺点，这些缺点即使在 MapReduce 最初大红大紫的阶段已经非常明显。其中许多缺点都与 MapReduce 完全限定的 Map→Shuffle→Reduce 编程模型相关；这个编程模型虽然简单，但它带来了一些缺点：

- 由于单个 MapReduce 作业并不能完成大量实际上的业务案例，因此许多定制的编排系统开始在 Google 公司内部出现，这些编排系统主要用于协调 MapReduce 作业的顺序。这些系统基本上都在解决同一类问题，即将多个 MapReduce 作业粘合在一起，创建一个解决复杂问题的数据管道。然而，这些编排系统都是 Google 各自团队独立开发的，相互之间也完全不兼容，是一类典型的重复造轮子案例。

- 更糟糕的是，由于 MapReduce 设计的 API 遵循严格结构，在很多情况下严格遵循 MapReduce 编程模型会导致作业运行效率低下。例如，一个团队可能会编写一个简单地过滤掉一些元素的 MapReduce，即，仅有 Map 阶段没有 Reduce 阶段的作业。这个作业下游紧接着另一个团队同样仅有 Map 阶段的作业，进行一些字段扩展和丰富（仍然带一个空的 Reduce 阶段作业）。第二个作业的输出最终可能会被第三个团队的 MapReduce 作业作为输入，第三个作业将对数据执行某些分组聚合。这个 Pipeline，实际上由一个合并 Map 阶段（译者注：前面两个 Map 合并为一个 Map），外加一个 Reduce 阶段即可完成业务逻辑，但实际上却需要编排三个完全独立的作业，每个作业通过 Shuffle 和 Output 两个步骤链接在一起。假设你希望保持代码的逻辑性和清洁性，于是你考虑将部分代码进行合并，但这个最终导致第三个问题。
- 为了优化 MapReduce 作业中的这些低效代码，工程师们开始引入手动优化，但不幸的是，这些优化会混淆 Pipeline 的简单逻辑，进而增加维护和调试成本。

Flume 通过提供可组合的高级 API 来描述数据处理流水线，从而解决了这些问题。这套设计理念同样也是 Beam 主要的抽象模型，即 PCollection 和 PTransform 概念，如图 10-9 所示。

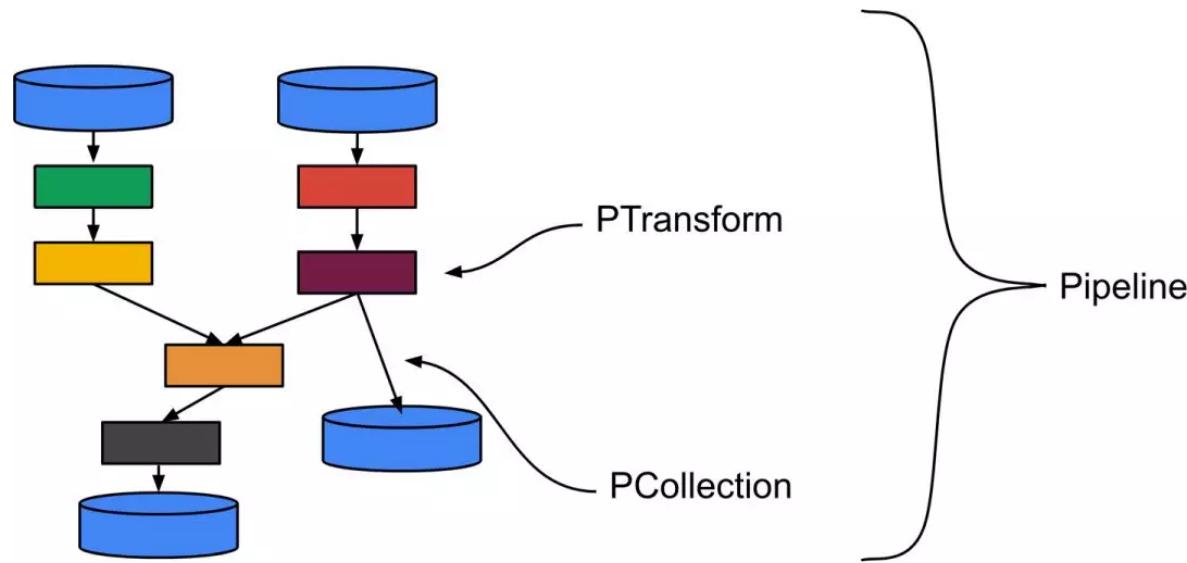


图 10-9 Flume 的高层抽象模型 (图片来源 : Frances Perry)

这些数据处理 Pipeline 在作业启动时将通过优化器生成，优化器将以最佳效率生成 MapReduce 作业，然后交由框架编排执行。整个编译执行原理图可以在图 10-10 中看到。



图 10-10 从逻辑管道到物理执行计划的优化

也许 Flume 在自动优化方面最重要的案例就是是合并（Reuven 在第 5 章中讨论了这个主题），其中两个逻辑上独立的阶段可以在同一个作业中顺序地（消费者 - 生产者融合）执行或者并行执行（兄弟融合），如图 10-11 所示。

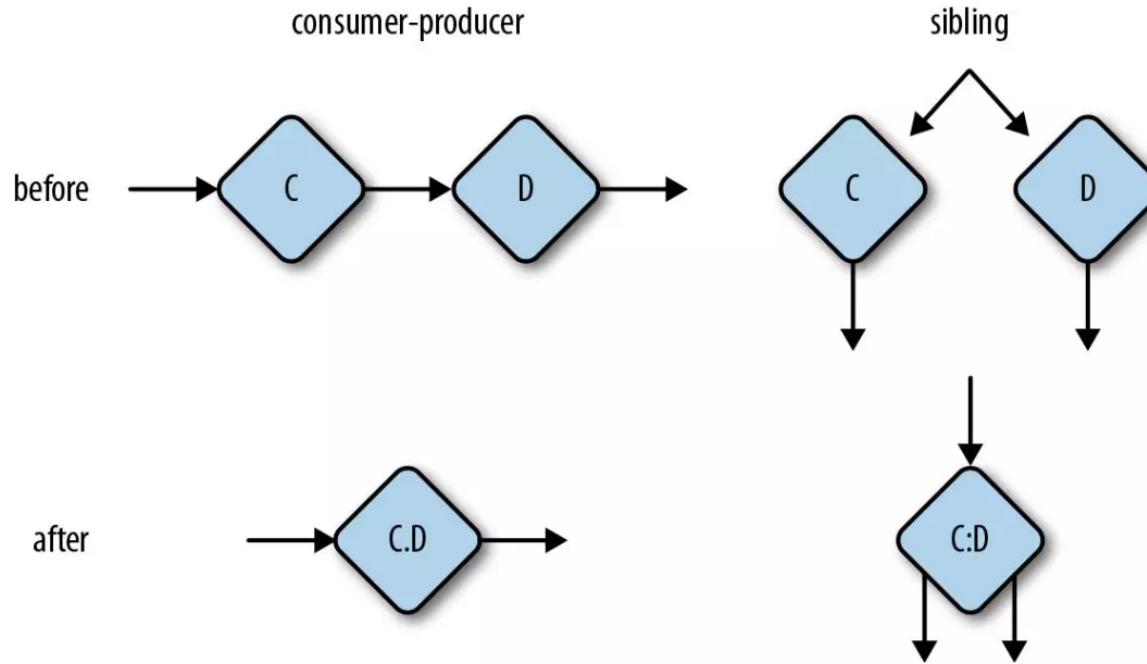


图 10-11 合并优化将顺序或并行操作(算子)组合在一起，到同一个操作(算子)。

将两个阶段融合在一起消除了序列化 / 反序列化和网络开销，这在处理大量数据的底层 Pipeline 中非常重要。

另一种类型的自动优化是 combiner lifting (见图 10-12)，当我们讨论增量合并时，我们已经在第 7 章中讨论了这些机制。combiner lifting 只是我们在该章讨论的多级组合逻辑的编译器自动优化：以求和操作为例，求和的合并逻辑本来应该运算在分组 (译者注: 即 Group-By) 操作后，由于优化的原因，被提前到在 group-by-key 之前做局部求和 (根据 group-by-key 的语义，经过 group-by-key 操作需要跨网络进行大量数据 Shuffle)。在出现数据热点情况下，将这个操作提前可以大大减少通过网络 Shuffle 的数据量，并且还可以在多台机器上分散掉最终聚合的机器负载。

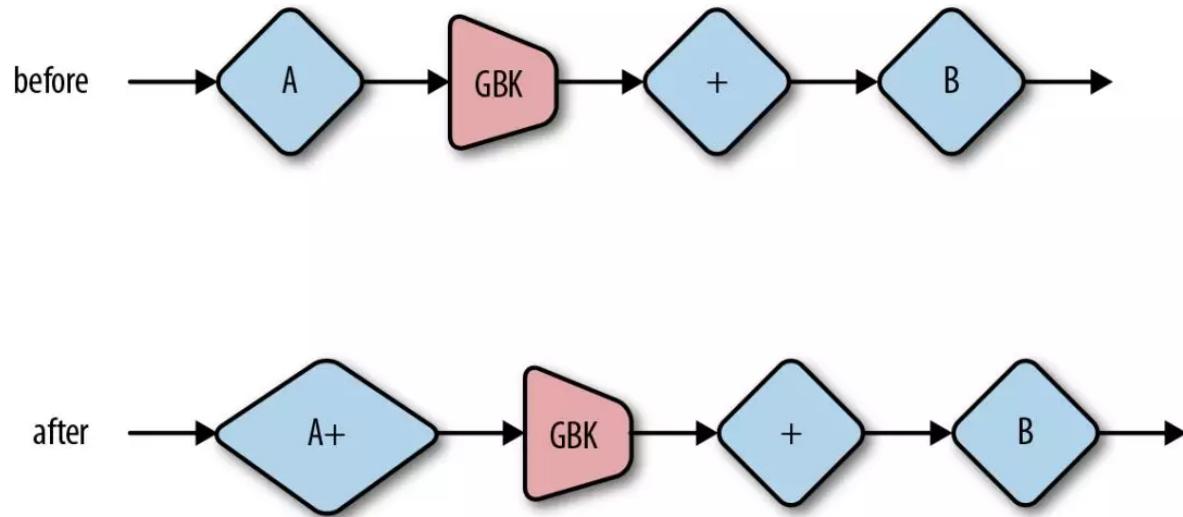


图 10-12: combiner lifting 在数据上游直接进行局部聚合后再发送给下游端进行二次聚合。

由于其更清晰的 API 定义和自动优化机制，在 2009 年初 Google 内部推出后 FlumeJava 立即受到巨大欢迎。之后，该团队发表了题为《Flume Java: Easy, Efficient Data-Parallel Pipelines》(<https://storage.googleapis.com/pub-tools-public-publication-data/pdf/35650.pdf>) 的论文（参见图 10-13），这篇论文本身就是一个很好的学习 FlumeJava 的资料。

FlumeJava: Easy, Efficient Data-Parallel Pipelines

Craig Chambers, Ashish Raniwala, Frances Perry,
Stephen Adams, Robert R. Henry,
Robert Bradshaw, Nathan Weizenbaum
Google, Inc.
`{chambers,raniwala,fjp,sra,rrh,robertwb,nweiz}@google.com`

Abstract

MapReduce and similar systems significantly ease the task of writing data-parallel code. However, many real-world computations require a pipeline of MapReduces, and programming and managing such pipelines can be difficult. We present FlumeJava, a Java library that makes it easy to develop, test, and run efficient data-parallel pipelines. At the core of the FlumeJava library are a couple of classes that represent immutable parallel collections, each supporting a modest number of operations for processing them in parallel. Parallel collections and their operations present a simple, high-level, uniform abstraction over different data representations and execution strategies. To enable parallel operations to run efficiently, FlumeJava defers their evaluation, instead internally constructing an execution plan dataflow graph. When the final results of the parallel operations are eventually needed, FlumeJava first optimizes the execution plan, and then executes the optimized operations on appropriate underlying primitives (e.g., MapReduces). The combination of high-level abstractions for parallel data and computation, deferred evaluation and optimization, and efficient parallel primitives yields an easy-to-use system that approaches the efficiency of hand-optimized pipelines. FlumeJava is in active use by hundreds of pipeline developers within Google.

MapReduce works well for computations that can be broken down into a map step, a shuffle step, and a reduce step, but for many real-world computations, a chain of MapReduce stages is required. Such data-parallel *pipelines* require additional coordination code to chain together the separate MapReduce stages, and require additional work to manage the creation and later deletion of the intermediate results between pipeline stages. The logical computation can become obscured by all these low-level coordination details, making it difficult for new developers to understand the computation. Moreover, the division of the pipeline into particular stages becomes “baked in” to the code and difficult to change later if the logical computation needs to evolve.

In this paper we present FlumeJava, a new system that aims to support the development of data-parallel pipelines. FlumeJava is a Java library centered around a few classes that represent *parallel collections*. Parallel collections support a modest number of *parallel operations* which are composed to implement data-parallel computations. An entire pipeline, or even multiple pipelines, can be implemented in a single Java program using the FlumeJava abstractions; there is no need to break up the logical computation into separate programs for each stage.

FlumeJava’s parallel collections abstract away the details of

图 10-13 FlumeJava 的论文

Flume C++ 版本很快于 2011 年发布。之后 2012 年初，Flume 被引入为 Google 的所有新工程师提供的 Noogler6 培训内容。MapReduce 框架于是最终被走向被替换的命运。

从那时起，Flume 已经迁移到不再使用 MapReduce 作为执行引擎；相反，Flume 底层基于一个名为 Dax 的内置自定义执行引擎。工作本身。不仅让 Flume 更加灵活选择执行计划而不必拘泥于 Map→Shuffle→Reduce MapReduce 的模型，Dax 还启用了新的优化，例如 Eugene Kirpi-chov 和

Malo Denielou 的《No shard left behind》博客文章
(<https://cloud.google.com/blog/products/gcp/no-shard-left-behind-dynamic-work-rebalancing-in-google-cloud-dataflow>) 中描述的动态负载均衡 (图 10-14) 。



No shard left behind: dynamic work rebalancing in Google Cloud Dataflow

Wednesday, May 18, 2016

Posted by Eugene Kirpichov, Senior Software Engineer and Malo Denielou, Software Engineer

Introduction

Today we continue the discussion of [Google Cloud Dataflow's](#) “zero-knobs” story. Previously we showcased Cloud Dataflow's capability for [Autoscaling](#), which dynamically adjusts the number of workers to the needs of your pipeline. In this post, we discuss *Dynamic Work Rebalancing* (known internally at Google as *Liquid Sharding*), which keeps the workers busy.

We'll show how this feature addresses the problem of stragglers (workers that take a long time to finish their part of the work, delaying completion of the job and keeping other resources idle), greatly improving performance and cost in many scenarios, and how it enables and works in concert with autoscaling.

The problem of stragglers in big data processing systems

In all major distributed data processing engines – from Google's original MapReduce, to Hadoop, to modern systems such as Spark, Flink and Cloud Dataflow – one of the key operations is Map, which applies a function to all elements of an input in parallel (called ParDo in the terminology of [Apache Beam \(incubating\)](#) programming model).

图 10-14 帖子《No shard left behind》

尽管那篇博客主要是基于 Google DataFlow 框架下讨论问题，但动态负载均衡（或液态分片，Google 内部更习惯这样叫）可以让部分已经完成工作的 Worker 能够从另外一些繁忙的 Worker 手中分配一些额外的工作。在 Job 运行过程中，通过不断的动态调整负载分配可以将系统运行效率趋近最优，这种算法将比传统方法下有经验工程师手工设置的初始参数性能更好。Flume 甚至为 Worker 池变化进行了适

配，一个拖慢整个作业进度的 Worker 会将其任务转移到其他更加高效的 Worker 上面进行执行。Flume 的这些优化手段，在 Google 内部为公司节省了大量资源。

最后一点，Flume 后来也被扩展为支持流语义。除 Dax 作为一个批处理系统引擎外，Flume 还扩展为能够在 MillWheel 流处理系统上执行作业（稍后讨论）。在 Google 内部，之前本书中讨论过的大多数高级流处理语义概念首先被整合到 Flume 中，然后才进入 Cloud Dataflow 并最终进入 Apache Beam。

总而言之，本节我们主要强调的是 Flume 产品给人引入高级管道概念，这使得能够让用户编写清晰易懂且自动优化的分布式大数据处理逻辑，从而让创建更大型更复杂的分布式大数据任务成为了可能，Flume 让我们业务代码在保持代码清晰逻辑干净的同时，自动具备编译器优化能力。

Storm

接下来是 Apache Storm（图 10-15），这是我们研究的第一个真正的流式系统。Storm 肯定不是业界使用最早的流式处理系统，但我认为这是整个行业真正广泛采用的第一个流式处理系统，因此我们在这里需要仔细研究一下。

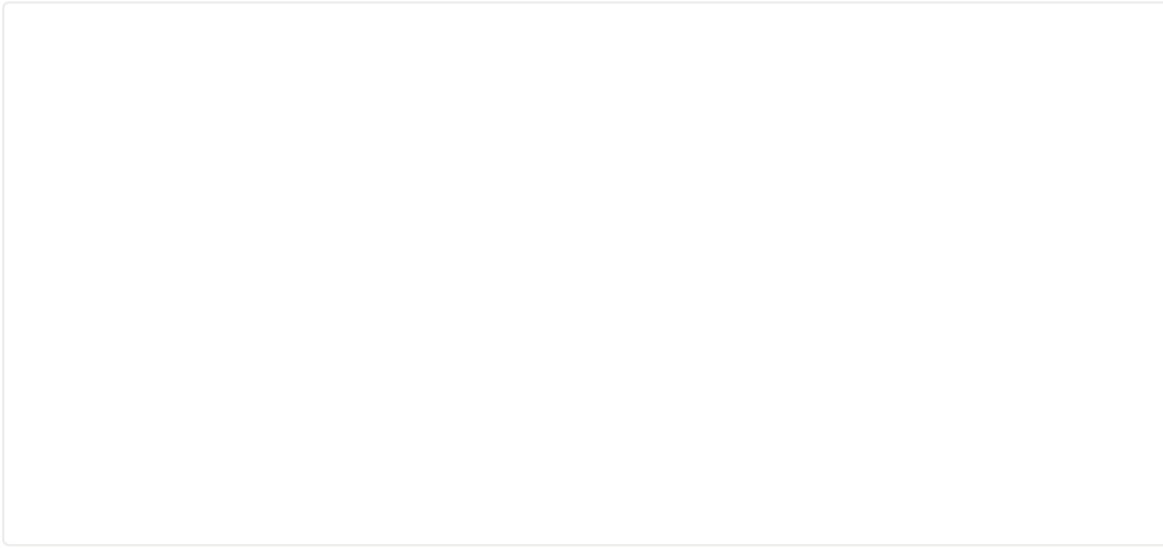


图 10-15 Storm 的时间轴

Storm 是 Nathan Marz 的心血结晶，Nathan Marz 后来在一篇题为《History of Apache Storm and lessons learned》的博客文章 (<http://nathanmarz.com/blog/history-of-apache-storm-and-lessons-learned.html>) 中记录了其创作历史（图 10-16）。这篇冗长的博客讲述了 BackType 这家创业公司一直在自己通过消息队列和自定义代码去处理 Twitter 信息流。Nathan 和十几年前 Google 里面设计 MapReduce 相关工程师有相同的认识：实际的业务处理的代码仅仅是系统代码很小一部分，如果有统一的流式实时处理框架负责处理各类分布式系统底层问题，那么基于之上构建我们的实时大数据处理将会轻松得多。基于此，Nathan 团队完成了 Storm 的设计和开发。

值得一提的是，Storm 的设计原则和其他系统大相径庭，Storm 更多考虑到实时流计算的处理时延而非数据的一致性保证。后者是其他大数据系统必备基础产品特征之一。Storm 针对每条流式数据进行计算处理，并提供至多一次或者至少一次的语义保证；同时不提供任何状态存储能力。相比于 Batch 批处理系统

能够提供一致性语义保证，Storm 系统能够提供更低的数据处理延迟。对于某些数据处理业务场景来说，这确实也是一个非常合理的取舍。

History of Apache Storm and lessons learned

MONDAY, OCTOBER 6, 2014

Apache Storm recently became a [top-level project](#), marking a huge milestone for the project and for me personally. It's crazy to think that four years ago Storm was nothing more than an idea in my head, and now it's a thriving project with a large community used by [a ton of companies](#). In this post I want to look back at how Storm got to this point and the lessons I learned along the way.



图 10-16 《History of Apache Storm and lessons learned》

不幸的是，人们很快就清楚地知道他们想要什么样的流式处理系统。他们不仅希望快速得到业务结果，同时希望系统具有低延迟和准确性，但仅凭 Storm 架构实际上不可能做到这一点。针对这个情况，Nathan 后面又提出了 Lambda 架构。

鉴于 Storm 的局限性，聪明的工程师结合弱一致语义的 Storm 流处理以及强一致语义的 Hadoop 批处理。前者产生了低延迟，但不精确的结果，而后者产生了高延迟，但精确的结果，双剑合璧，整合两套系统整体提供的低延迟但最终一致的输出结果。我们在第 1 章中了解到，Lambda 架构是 Marz 的另一个创意，详见他的文章《“如何击败 CAP 定理”》(<http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html>)（图 10-17）。

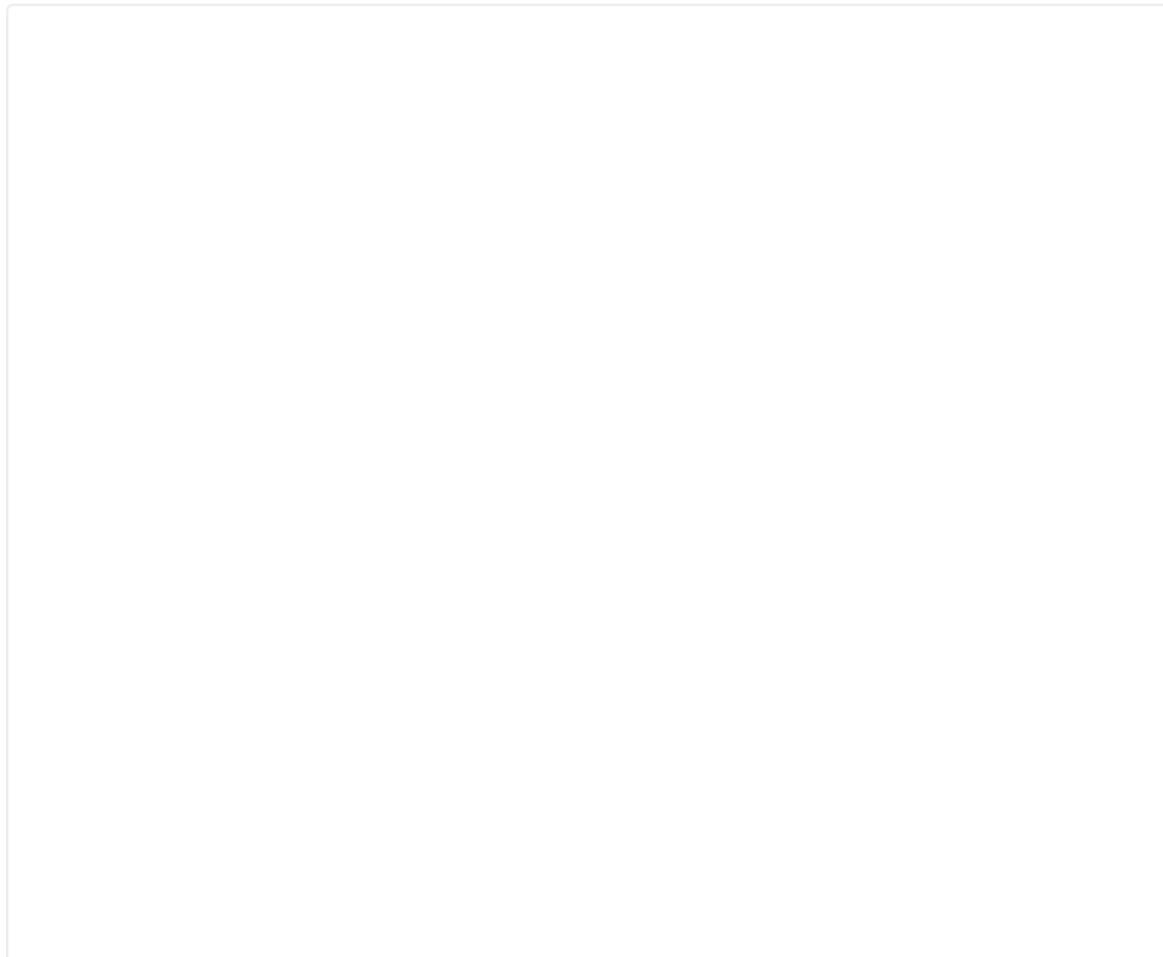


图 10-17 《How to beat the CAP theorem》

我已经花了相当多的时间来分析 Lambda 架构的缺点，以至于我不会在这里啰嗦这些问题。但我要重申一下：尽管它带来了大量成本问题，Lambda 架构当前还是非常受欢迎，仅仅是因为它满足了许多企业一个关键需求：系统提供低延迟但不准确的数据，后续通过批处理系统纠正之前数据，最终给出一致性的结果。从流处理系统演变的角度来看，Storm 确实为普罗大众带来低延迟的流式实时数据处理能力。然而，它是以牺牲数据强一致性为代价的，这反过来又带来了 Lambda 架构的兴起，导致接下来多年基于两套系统架构之上的数据处理带来无尽的麻烦和成本。

撇开其他问题先不说，Storm 是行业首次大规模尝试低延迟数据处理的系统，其影响反映在当前线上大量部署和应用各类流式处理系统。在我们要放下 Storm 开始聊其他系统之前，我觉得还是很有必要去说说 Heron 这个系统。在 2015 年，Twitter 作为 Storm 项目孵化公司以及世界上已知最大的 Storm 用户，突然宣布放弃 Storm 引擎，宣称正在研发另外一套称之为 Heron 的流式处理框架。Heron 旨在解决困扰 Storm 的一系列性能和维护问题，同时向 Storm 保持 API 兼容，详见题为《Twitter Heron : Stream Processing at scale》的论文（<https://www.semanticscholar.org/paper/Twitter-Heron%3A-Stream-Processing-at-Scale-Kulkarni-Bhagat/e847c3ec130da57328db79a7fea794b07dbccdd9>）（图 10-18）。



图 10-18 Heron 的论文

Heron 本身也是开源产品（但开源不在 Apache 项目中）。鉴于 Storm 仍然在社区中持续发展，现在又冒出一套和 Storm 竞争的软件，最终两边系统鹿死谁手，我们只能拭目以待了。

Spark

继续走起，我们现在来到 Apache Spark (图 10-19)。再次，我又将大量简化 Spark 系统对行业的总体影响探讨，仅仅关注我们的流处理领域部分。

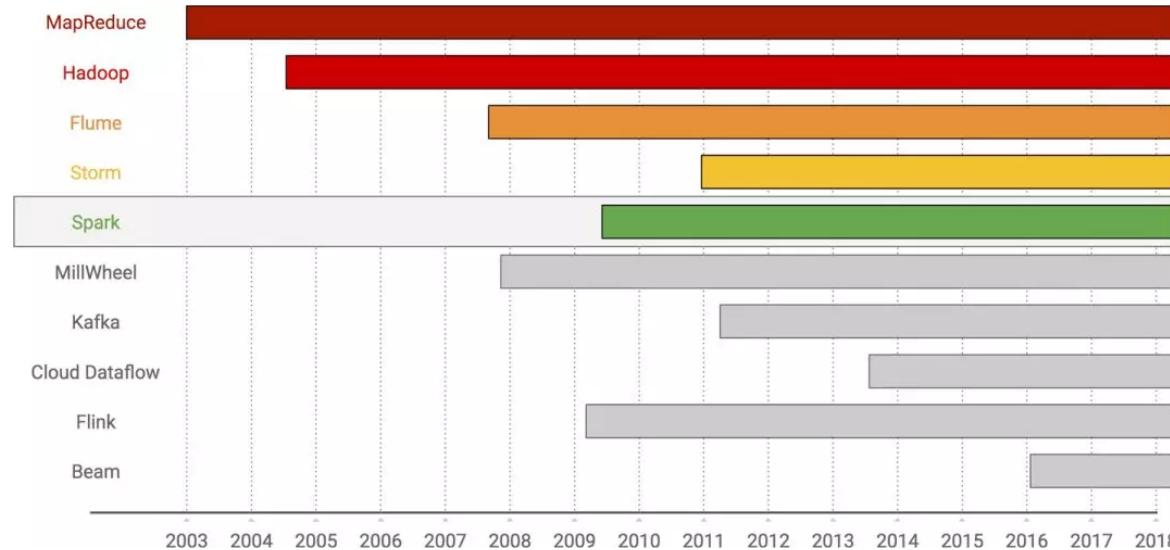


图 10-19 Spark 的时间轴

Spark 在 2009 年左右诞生于加州大学伯克利分校的著名 AMPLab。最初推动 Spark 成名的原因是它能够经常在内存执行大量的计算工作，直到作业的最后一步才写入磁盘。工程师通过弹性分布式数据集 (RDD) 理念实现了这一目标，在底层 Pipeline 中能够获取每个阶段数据结果的所有派生关系，并且允许在机器故障时根据需要重新计算中间结果，当然，这些都基于一些假设 a) 输入是总是可重放的，b) 计算是确定性的。对于许多案例来说，这些先决条件是真实的，或者看上去足够真实，至少用户确实在

Spark 享受到了巨大的性能提升。从那时起，Spark 逐渐建立起其作为 Hadoop 事实上的继任产品定位。

在 Spark 创建几年后，当时 AMPLab 的研究生 Tathagata Das 开始意识到：嘿，我们有这个快速的批处理引擎，如果我们将多个批次的任务串接起来，用它能否来处理流数据？于是乎，Spark Streaming 诞生了。

关于 Spark Streaming 的真正精彩之处在于：强大的批处理引擎解决了太多底层麻烦的问题，如果基于此构建流式处理引擎则整个流处理系统将简单很多，于是世界又多一个流处理引擎，而且是可以独自提供一致性语义保障的流式处理系统。换句话说，给定正确的用例，你可以不用 Lambda 架构系统直接使用 Spark Streaming 即可满足数据一致性需求。为 Spark Streaming 手工点赞！

这里的一个主要问题是“正确的用例”部分。早期版本的 Spark Streaming (1.x 版本)的一大缺点是它仅支持特定的流处理语义：即，处理时间窗口。因此，任何需要使用事件时间，需要处理延迟数据等等案例都无法让用户使用 Spark 开箱即用解决业务。这意味着 Spark Streaming 最适合于有序数据或事件时间无关的计算。而且，正如我在本书中重申的那样，在处理当今常见的大规模、以用户为中心的数据集时，这些先决条件看上去并不是那么常见。

围绕 Spark Streaming 的另一个有趣的争议是“microbatch 和 true streaming”争论。由于 Spark Streaming 建立在批处理引擎的重复运行的基础之上，因此批评者声称 Spark Streaming 不是真正的流式引擎，因为整个系统的处理基于全局的数据切分规则。这个或多或少是实情。尽管流处理引擎几乎总是为了吞吐量而使用某种批处理或者类似的加大吞吐的系统策略，但它们可以灵活地在更精细的级别上进行处理，一直可以细化到某个 key。但基于微批处理模型的系统在基于全局切分方式处理数据包，这意味着

同时具备低延迟和高吞吐是不可能的。确实我们看到许多基准测试表明这说法或多或少有点正确。当然，作业能够做到几分钟或几秒钟的延迟已经相当不错了，实际上生产中很少有用例需要严格数据正确性和低延迟保证。所以从某种意义上说，Spark 瞄准最初目标客户群体打法是非常到位的，因为大多数业务场景均属于这一类。但这并未阻止其竞争对手将此作为该平台的巨大劣势。就个人而言，在大多数情况下，我认为这只是一个很小问题。

撇开缺点不说，Spark Streaming 是流处理的分水岭：第一个广泛使用的大规模流处理引擎，它也可以提供批处理系统的正确性保证。当然，正如前面提到的，流式系统只是 Spark 整体成功故事的一小部分，Spark 在迭代处理和机器学习领域做出了重要贡献，其原生 SQL 集成以及上述快如闪电般的内存计算，都是非常值得大书特书的产品特性。

如果您想了解有关原始 Spark 1.x 架构细节的更多信息，我强烈推荐 Matei Zaharia 关于该主题的论文《“An Architecture for Fast and General Data Processing on Large Clusters”》（图 10-20）。这是 113 页的 Spark 核心讲解论文，非常值得一读。



图 10-20 Spark 的学位论文

时至今日，Spark 的 2.x 版本极大地扩展了 Spark Streaming 的语义功能，其中已经包含了本书中描述流式处理模型的许多部分，同时试图简化一些更复杂的设计。Spark 甚至推出了一种全新的、真正面向流

式处理的架构，用以规避掉微批架构的种种问题。但是曾经，当 Spark 第一次出现时，它带来的最重要贡献是它是第一个公开可用的流处理引擎，具有数据处理的强一致性语义，尽管这个特性只能用在有序数据或使用处理时间计算的场景。

MillWheel

接下来我们讨论 MillWheel，这是我在 2008 年加入 Google 后的花 20% 时间兼职参与的项目，后来在 2010 年全职加入该团队（图 10-21）。

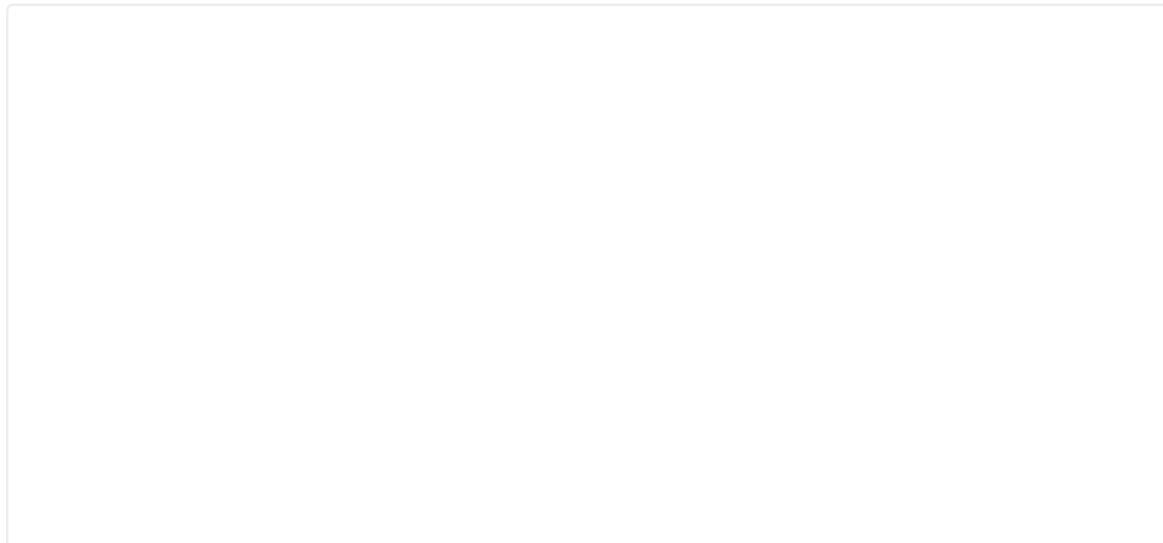


图 10-21 MillWheel 时间表

MillWheel 是 Google 最早的通用流处理架构，该项目由 Paul Nordstrom 在 Google 西雅图办事处开业时发起。 MillWheel 在 Google 内的成功与长期以来一直致力于为无序数据提供低延迟，强一致的处理能力不无关系。在本书的讲解中，我们已经多次分别讨论了促使 MillWheel 成为一款成功产品的方方面面。

- 第五章，Reuven 详细讨论过数据精准一次的语义保证。精准一次的语义保证对于正确性至关重要。
- 第七章，我们研究了状态持久化，这为在不那么靠谱的普通硬件上执行的长时间数据处理业务并且需要保证正确性奠定了基础。
- 第三章，Slava 讨论了 Watermark。Watermark 为处理无序数据提供了基础。
- 第七章，我们研究了持久性计时器，它们提供了 Watermark 与业务逻辑之间的某些关联特性。

有点令人惊讶的是，MillWheel 项目最开始并未关注数据正确性。保罗最初的想法更接近于 Storm 的设计理论：具有弱一致性的低延迟数据处理。这是最初的 MillWheel 客户，一个关于基于用户搜索数据构建会话和另一个对搜索查询执行异常检测（来自 MillWheel 论文的 Zeitgeist 示例），这两家客户迫使项目走向了正确的方向。两者都非常需要强一致的数据结果：会话用于推断用户行为，异常检测用于推断搜索查询的趋势；如果他们提供的数据不靠谱，两者效果都会显着下降。最终，幸运的是，MillWheel 的设计被客户需求导向追求数据强一致性的结果。

支持乱序数据处理，这是现代流式处理系统的另一个核心功能。这个核心功能通常也被认为是被 MillWheel 引入到流式处理领域，和数据准确性一样，这个功能也是被客户需求推动最终加入到我们系统。Zeitgeist 项目的大数据处理过程，通常被我们拿来用作一个真正的流式处理案例来讨论。Zeitgeist 项目希望检测识别搜索查询流量中的异常，并且需要捕获异常流量。对于这个大数据项目数据消费者来

说，流计算将所有计算结果产出并让用户轮询所有 key 用来识别异常显然不太现实，数据用户要求系统直接计算某个 key 出现异常的数据结果，而不需要上层再来轮询。对于异常峰值（即查询流量的增加），这还相对来说比较简单好解决：当给定查询的计数超过查询的预期值时，系统发出异常信号。但是对于异常下降（即查询流量减少），问题有点棘手。仅仅看到给定搜索词的查询数量减少是不够的，因为在任何时间段内，计算结果总是从零开始。在这些情况下你必须确保你的数据输入真的能够代表当前这段时间真实业务流量，然后才将计算结果和预设模型进行比较。

真正的流式处理

“真正的流式处理用例”需要一些额外解释。流式系统的一个新的演化趋势是，舍弃掉部分产品需求以简化编程模型，从而使整个系统简单易用。例如，在撰写本文时，Spark Structured Streaming 和 Apache Kafka Streams 都将系统提供的功能限制在第 8 章中称为“物化视图语义”范围内，本质上对最终一致性的输出表不停做数据更新。当您想要将上述输出表作为结果查询使用时，物化视图语义非常匹配你的需求：任何时候我们只需查找该表中的值并且（译者注：尽管结果数据一直在不停被更新和改变）以当前查询时间请求到查询结果就是最新的结果。但在一些需要真正流式处理的场景，例如异常检测，上述物化视图并不能够很好地解决这类问题。

接下来我们会讨论到，异常检测的某些需求使其不适合纯物化视图语义（即，依次针对单条记录处理），特别当需要完整的数据集才能够识别业务异常，而这些异常恰好是由于数据的缺失或者不完整导致的。另外，不停轮询结果表以查看是否有异常其实并不是一个扩展性很好的办法。真正的流式用户场景是推动 watermark 等功能的原始需求来源。（Watermark 所代表的时间有先有后，我们需要最低的 Watermark 追踪数据的完整性，而最高的 Watermark 在数据时间发生倾斜时候非常容易导致丢数据的情况发生，类似 Spark Structured Streaming 的用法）。省略类似 Watermark 等功能的系统看上

去简单不少，但换来代价是功能受限。在很多情况下，这些功能实际上有非常重要的业务价值。但如果这样的系统声称这些简化的功能会带来系统更多的普适性，不要听他们忽悠。试问一句，功能需求大量被砍掉，如何保证系统的普适性呢？

Zeitgeist 项目首先尝试通过在计算逻辑之前插入处理时间的延迟数值来解决数据延迟问题。当数据按顺序到达时，这个思路处理逻辑正常。但业务人员随后发现数据有时可能会延迟很大，从而导致数据无序进入流式处理系统。一旦出现这个情况，系统仅仅采用处理时间的延迟是不够的，因为底层数据处理会因为数据乱序原因被错误判断为异常。最终，我们需要一种等待数据到齐的机制。

之后 Watermark 被设计出来用以解决数据乱序的问题。正如 Slava 在第 3 章中所描述的那样，基本思想是跟踪系统输入数据的当前进度，对于每个给定的数据源，构建一个数据输入进度用来表征输入数据的完整性。对于一些简单的数据源，例如一个带分区的 Kafka Topic，每个 Topic 下属的分区被写入的是业务时间持续递增的数据（例如通过 Web 前端实时记录的日志事件），这种情况下我们可以计算产生一个非常完美的 Watermark。但对于一些非常复杂的数据输入，例如动态的输入日志集，一个启发式算法可能是我们能够设计出来最能解决业务问题的 Watermark 生成算法了。但无论哪种方式，Watermark 都是解决输入事件完整性最佳方式。之前我们尝试使用处理时间来解决事件输入完整性，有点驴头不及马嘴的感觉。

得益于客户的需求推动，MillWheel 最终成为能够支持无序数据的强大流处理引擎。因此，题为《MillWheel: Fault-Tolerant Stream Processing at Internet Scale》（图 10-22）的论文花费大部分时间来讨论在这样的系统中提供正确性的各种问题，一致性保证、Watermark。如果您对这个主题感兴趣，那值得花时间去读读这篇论文。

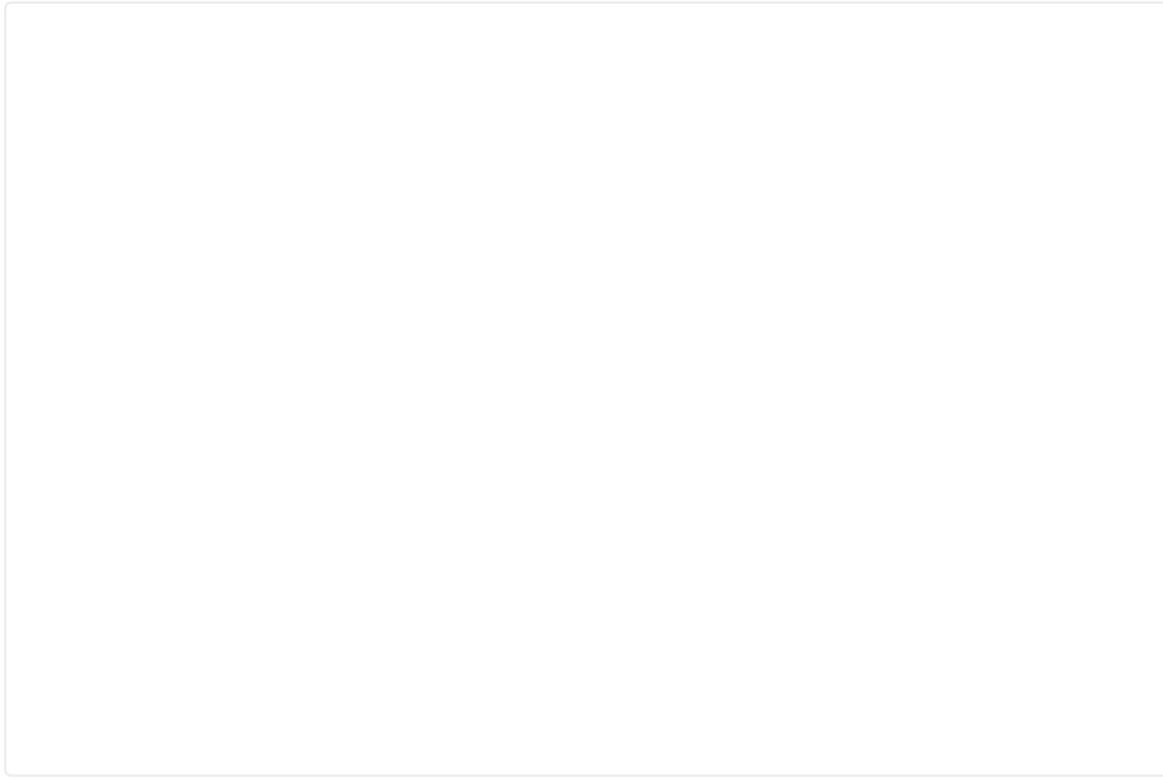


图 10-22 MillWheel 论文

MillWheel 论文发表后不久，MillWheel 就成为 Flume 底层提供支撑的流式处理引擎，我们称之为 Streaming Flume。今天在谷歌内部，MillWheel 被下一代理论更为领先的系统所替换：Windmill（这套系统同时也为 DataFlow 提供了执行引擎），这是一套基于 MillWheel 之上，博采众家之长的大数据处理系统，包括提供更好的调度和分发策略、更清晰的框架和业务代码解耦。

MillWheel 给我们带来最大的价值是之前列出的四个概念（数据精确一次性处理，持久化的状态存储，Watermark，持久定时器）为流式计算提供了工业级生产保障：即使在不可靠的商用硬件上，也可以对无序数据进行稳定的、低延迟的处理。

Kafka

我们开始讨论 Kafka（图 10-23）。Kafka 在本章讨论的系统中是独一无二的，因为它不是数据计算框架，而是数据传输和存储的工具。但是，毫无疑问，Kafka 在我们正在讨论的所有系统中扮演了推动流处理的最有影响力的角色之一。

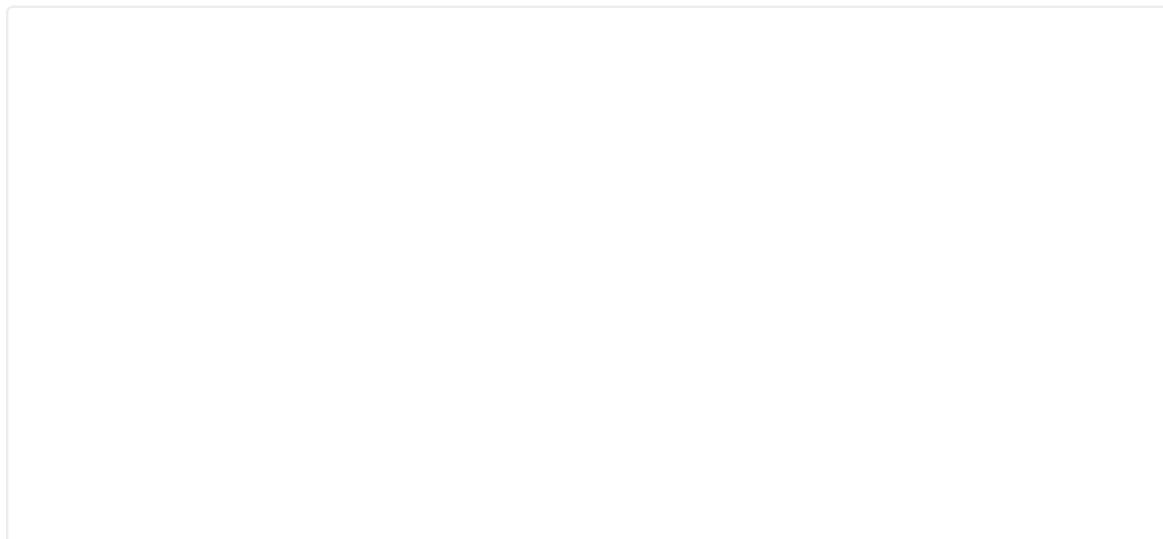


图 10-23 Kafka 的时间轴

如果你不熟悉它，我们可以简单描述为：Kafka 本质上是一个持久的流式数据传输和存储工具，底层系统实现为一组带有分区结构的日志型存储。它最初是由 Neha Narkhede 和 Jay Kreps 等业界大牛在 LinkedIn 公司内部开发的，其卓越的特性有：

- 提供一个干净的持久性模型，让大家在流式处理领域里面可以享受到批处理的产品特性，例如持久化、可重放。
- 在生产者和消费者之间提供弹性隔离。
- 我们在第 6 章中讨论过的流和表之间的关系，揭示了思考数据处理的基本方式，同时还提供了和数据库打通的思路和概念。
- 来自于上述所有方面的影响，不仅让 Kafka 成为整个行业中大多数流处理系统的基础，而且还促进了流处理数据库和微服务运动。

在这些特性中，有两个对我来说最为突出。第一个是流数据的持久化和可重放性的应用。在 Kafka 之前，大多数流处理系统使用某种临时、短暂的消息系统，如 Rabbit MQ 甚至是普通的 TCP 套接字来发送数据。数据处理的一致性往往通过生产者数据冗余备份来实现（即，如果下游数据消费者出现故障，则上游生产者将数据进行重新发送），但是上游数据的备份通常也是临时保存一下。大多数系统设计完全忽略在开发和测试中需要重新拉取数据重新计算的需求。但 Kafka 的出现改变了这一切。从数据库持久日志概念得到启发并将其应用于流处理领域，Kafka 让我们享受到了如同 Batch 数据源一样的安全性和可靠性。凭借持久化和可重放的特点，流计算在健壮性和可靠性上面又迈出关键的一步，为后续替代批处理系统打下基础。

作为一个流式系统开发人员，Kafka 的持久化和可重放功能对业界产生一个更有意思的变化就是：当今天大量流处理引擎依赖源头数据可重放来提供端到端精确一次的计算保障。可重放的特点是 Apex , Flink , Kafka Streams , Spark 和 Storm 的端到端精确一次保证的基础。当以精确一次模式执行时，每个系统都假设 / 要求输入数据源能够重放之前的部分数据（从最近 Checkpoint 到故障发生时的数据）。当流式处理系统与不具备重放能力的输入源一起使用时（哪怕是源头数据能够保证可靠的一致性数据投递，但不能提供重放功能），这种情况下无法保证端到端的完全一次语义。这种对可重放（以及持久化等其他特点）的广泛依赖是 Kafka 在整个行业中产生巨大影响的间接证明。

Kafka 系统中第二个值得注意的重点是流和表理论的普及。我们花了整个第 6 章以及第 8 章、第 9 章来讨论流和表，可以说流和表构成了数据处理的基础，无论是 MapReduce 及其演化系统，SQL 数据库系统，还是其他分支的数据处理系统。并不是所有的数据处理方法都直接基于流或者表来进行抽象，但从概念或者理论上说，表和流的理论就是这些系统的运作方式。作为这些系统的用户和开发人员，理解我们所有系统构建的核心基础概念意义重大。我们都非常感谢 Kafka 社区的开发者，他们帮助我们更广泛更加深入地了解到批流理论。

如果您想了解更多关于 Kafka 及其理论核心，JackKreps 的《I ❤️ Logs》（ O'Reilly; 图 10-24 ）是一个很好的学习资料。另外，正如第 6 章中引用的那样，Kreps 和 Martin Kleppmann 有两篇文章（图 10-25 ），我强烈建议您阅读一下关于流和表相关理论。

The O'Reilly logo, featuring the word "OREILLY" in white capital letters on a red background, enclosed in a white rectangular border.



I ❤️ Logs

EVENT DATA, STREAM PROCESSING, AND DATA INTEGRATION

Jay Kreps

图 10-24 《I ❤️ Logs》

Kafka 为流处理领域做出了巨大贡献，可以说比其他任何单一系统都要多。特别是，对输入和输出流的持久性和可重放的设计，帮助将流计算从近似工具的小众领域发展到在大数据领域妇孺皆知的程度起了很大作用。此外，Kafka 社区推广的流和表理论对于数据处理引发了我们深入思考。

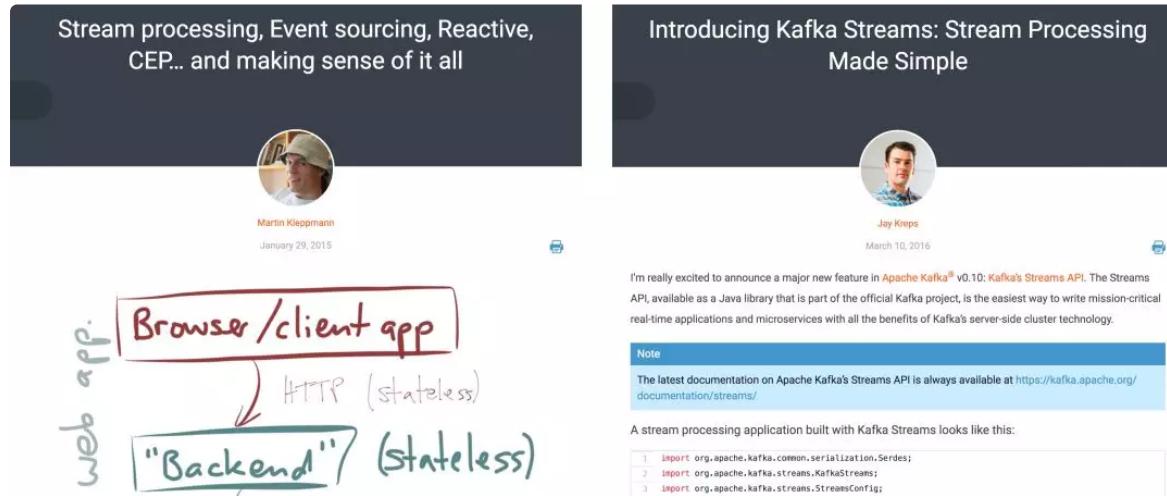


图10-25 Martin 的帖子 (左边) 以及 Jay 的帖子 (右边)

DataFlow

Cloud Dataflow (图 10-26) 是 Google 完全托管的、基于云架构的数据处理服务。Dataflow 于 2015 年 8 月推向全球。DataFlow 将 MapReduce , Flume 和 MillWheel 的十多年经验融入其中，并将其打包成 Serverless 的云体验。



图 10-26 Google DataFlow 的时间轴

虽然 Google 的 Dataflow 的 Serverless 特点可能是从系统角度来看最具技术挑战性以及有别于其他云厂商产品的重要因素，但我想在此讨论主要是其批流统一的编程模型。编程模型包括我们在本书的大部分内容中所讨论的转换，窗口，水印，触发器和聚合计算。当然，所有这些讨论都包含了思考问题的 what、where、when、how。

DataFlow 模型首先诞生于 Flume，因为我们希望将 MillWheel 中强大的无序数据计算能力整合到 Flume 提供的更高级别的编程模型中。这个方式可以让 Google 员工在内部使用 Flume 进行统一的批处理和流处理编程。

关于统一模型的核心关键思考在于，尽管在当时我们也没有深刻意识到，批流处理模型本质上没有区别：仅仅是在表和流的处理上有些小变化而已。正如我们在第 6 章中所讨论到的，主要的区别仅仅是在将表上增量的变化转换为流，其他一切在概念上是相同的。通过利用批处理和流处理两者大量的共性需求，可以提供一套引擎，适配于两套不同处理方式，这让流计算系统更加易于使用。

除了利用批处理和流处理之间的系统共性之外，我们还仔细查看了多年来我们在 Google 中遇到的各种案例，并使用这些案例来研究统一模型下系统各个部分。我们研究主要内容如下：

- 未对齐的事件时间窗口（如会话窗口），能够简明地表达这类复杂的分析，同时亦能处理乱序数据。
- 自定义窗口支持，系统内置窗口很少适合所有业务场景，需要提供给用户自定义窗口的能力。
- 灵活的触发和统计模式，能够满足正确性，延迟，成本的各项业务需求。

- 使用 Watermark 来推断输入数据的完整性，这对于异常检测等用例至关重要，其中异常检测逻辑会根据是否缺少数据做出异常判断。
- 底层执行环境的逻辑抽象，无论是批处理，微批处理还是流式处理，都可以在执行引擎中提供灵活的选择，并避免系统级别的参数设置（例如微批量大小）进入逻辑 API。

总之，这些平衡了灵活性，正确性，延迟和成本之间的关系，将 DataFlow 的模型应用于大量用户业务案例之中。

考虑到我们之前整本书都在讨论 DataFlow 和 Beam 模型的各类问题，我在此处重新给大家讲述这些概念纯属多此一举。但是，如果你正在寻找稍微更具学术性的内容以及一些应用案例，我推荐你看下 2015 年发表的《DataFlow 论文..》（图 10-27）。

The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing

Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak,
Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills,
Frances Perry, Eric Schmidt, Sam Whittle
Google

{takidau, robertwb, chambers, chernyak, rfernand,
relax, sgmc, mills, fjp, cloude, samuelw}@google.com

ABSTRACT

Unbounded, unordered, global-scale datasets are increasingly common in day-to-day business (e.g. Web logs, mobile usage statistics, and sensor networks). At the same time, consumers of these datasets have evolved sophisticated requirements, such as event-time ordering and windowing by features of the data themselves, in addition to an insatiable hunger for faster answers. Meanwhile, practicality dictates that one can never fully optimize along all dimensions of correctness, latency, and cost for these types of input. As a result, data processing practitioners are left with the quandary of how to reconcile the tensions between these seemingly competing propositions, often resulting in disparate implementations and systems.

We propose that a fundamental shift of approach is necessary to deal with these evolved requirements in modern data processing. We as a field must stop trying to groom unbounded datasets into finite pools of information that eventually become complete, and instead live and breathe under the assumption that we will never know if or when we have seen all of our data, only that new data will arrive, old data may be retracted, and the only way to make this problem tractable is via principled abstractions that allow the practitioner the choice of appropriate tradeoffs along the axes of interest: correctness, latency, and cost.

1. INTRODUCTION

Modern data processing is a complex and exciting field. From the scale enabled by MapReduce [16] and its successors (e.g Hadoop [4], Pig [18], Hive [29], Spark [33]), to the vast body of work on streaming within the SQL community (e.g. query systems [1, 14, 15], windowing [22], data streams [24], time domains [28], semantic models [9]), to the more recent forays in low-latency processing such as Spark Streaming [34], MillWheel, and Storm [5], modern consumers of data wield remarkable amounts of power in shaping and taming massive-scale disorder into organized structures with far greater value. Yet, existing models and systems still fall short in a number of common use cases.

Consider an initial example: a streaming video provider wants to monetize their content by displaying video ads and billing advertisers for the amount of advertising watched. The platform supports online and offline views for content and ads. The video provider wants to know how much to bill each advertiser each day, as well as aggregate statistics about the videos and ads. In addition, they want to efficiently run offline experiments over large swaths of historical data.

Advertisers/content providers want to know how often and for how long their videos are being watched, with which content/ads, and by which demographic groups. They also want to know how much they are being charged/paid. They

图 10-27 DataFlow 的论文

DataFlow 还有不少可以大书特书的功能特点，但在这章内容构成来看，我认为 DataFlow 最重要的是构建了一套批流统一的大数据处理模型。DataFlow 为我们提供了一套全面的处理无界且无序数据集的能力，同时这套系统很好的平衡了正确性、延迟、成本之间的相互关系。

Flink

Flink (图 10-28) 在 2015 年突然出现在大数据舞台，然后似乎在一夜之间从一个无人所知的系统迅速转变为人人皆知的流式处理引擎。

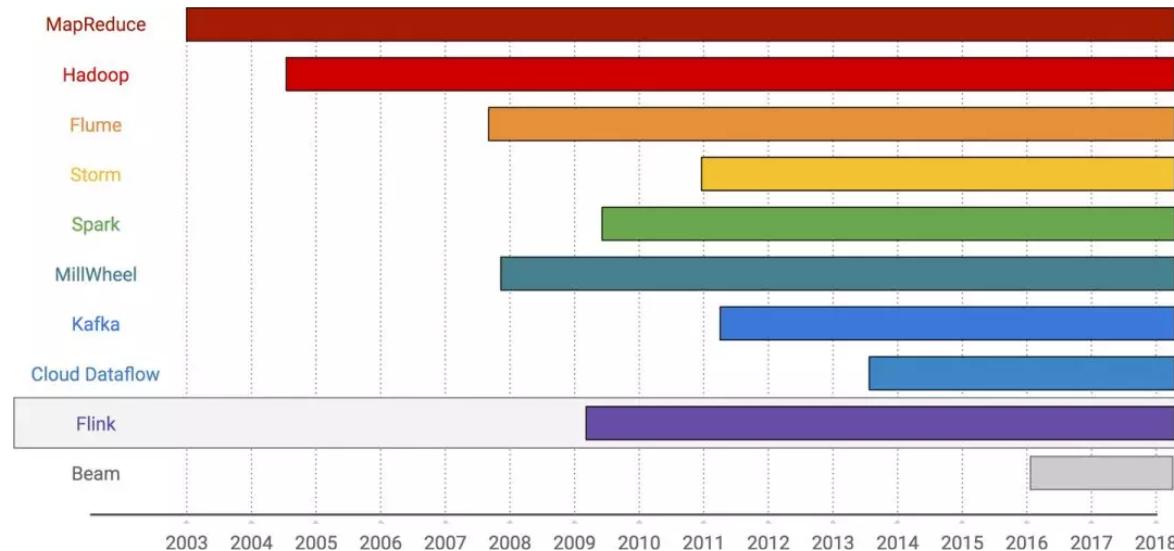


图 10-28 Flink 的时间轴

在我看来，Flink 崛起有两个主要原因：

- 采用 Dataflow/Beam 编程模型，使其成为完备语义功能的开源流式处理系统。

- 其高效的快照实现方式，源自 Chandy 和 Lamport 的原始论文《“Distributed Snapshots: Determining Global States of Distributed Systems”》的研究，这为其提供了正确性所需的强一致性保证。

Reuven 在第 5 章中简要介绍了 Flink 的一致性机制，这里在重申一下，其基本思想是在系统中的 Worker 之间沿着数据传播路径上产生周期性 Barrier。这些 Barrier 充当了在不同 Worker 之间传输数据时的对齐机制。当一个 Worker 在其所有上游算子输入来源（即来自其所有上游一层的 Worker）上接收到全部 Barrier 时，Worker 会将当前所有 key 对应的状态写入一个持久化存储。这个过程意味着将这个 Barrier 之前的所有数据都做了持久化。



图 10-29 Chandy-Lamport 快照

通过调整 Barrier 的生成频率，可以间接调整 Checkpoint 的执行频率，从而降低时延并最终获取更高的吞吐（其原因是做 Checkpoint 过程中涉及到对外进行持久化数据，因此会有一定的 IO 导致延时）。

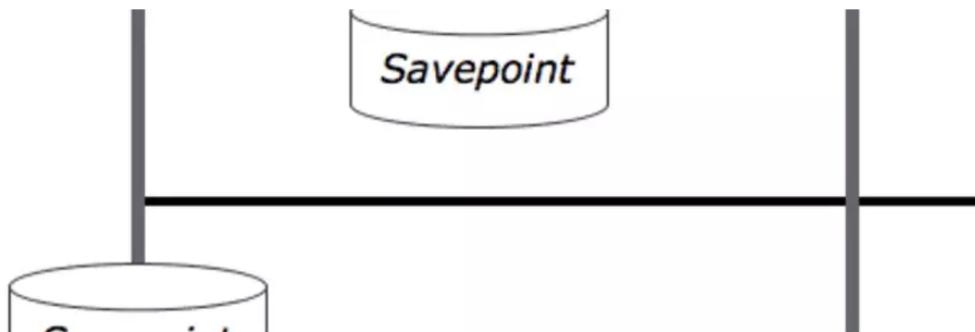
Flink 既能够支持精确一次的语义处理保证，同时又能够提供支持事件时间的处理能力，这让 Flink 获得的巨大成功。接着，Jamie Grier 发表他的题为 “《Extending the Yahoo! Streaming Benchmark》”（图 10-30）的文章，文章中描述了 Flink 性能具体的测试数据。在那篇文章中，杰米描述了两个令人印象深刻的特点：

1. 构建一个用于测试的 Flink 数据管道，其拥有比 Twitter Storm 更高的准确性（归功于 Flink 的强一次性语义），但成本却降到了 1%。

图 10-30. 《Extending the Yahoo! Streaming Benchmark》

2. Flink 在精确一次的处理语义参数设定下，仍然达到 Storm 的 7.5 倍吞吐量（而且，Storm 还不具备精确一次的处理语义）。此外，由于网络被打满导致 Flink 的性能受到限制；进一步消除网络瓶颈后 Flink 的吞吐量几乎达到 Storm 的 40 倍。

从那时起，许多其他流式处理项目（特别是 Storm 和 Apex）都采用了类似算法的数据处理一致性机制。



Savepoints: Turning Back Time

October 14, 2016 - Flink Features, Resources

Fabian Hueske and Michael Winters



This post is the first in a series where the data Artisans team will highlight some of Apache Flink's® core features. By Fabian Hueske (@fhueske) and Mike Winters (@wints)

Stream processing is commonly associated with 'data in motion', powering systems that make sense of and respond to data in nearly the same instant it's created. The most frequently discussed streaming topics, such as latency and throughput or watermarks and handling of late data, focus on the present rather than the past.

In reality, though, there are a number of cases where you'll need to reprocess data that your streaming application has already processed before. Some examples include:

- Deployment of a new version of your application with a new feature, a bug fix, or a better machine learning model
- A/B testing different versions of an application using the same source data streams, starting the

图 10-31 《Savepoints: Turning Back Time》

通过快照机制，Flink 获得了端到端数据一致性。Flink 更进了一步，利用其快照的全局特性，提供了从过去的任何一点重启整个管道的能力，这一功能称为 SavePoint（在 Fabian Hueske 和 Michael Winters 的帖子 [《Savepoints: Turning Back Time》 (<https://data-artisans.com/blog/turning-back-time-savepoints>)] 中有所描述，[图 10-31]）。Savepoints 功能参考了 Kafka 应用于流式传输层的持久化和可重放特性，并将其扩展应用到整个底层 Pipeline。流式处理仍然遗留大量开放性问题有待优化和提升，但 Flink 的 Savepoints 功能是朝着正确方向迈出的第一步，也是整个行业非常有特点的一步。如果您有兴趣了解有关 Flink 快照和保存点的系统构造的更多信息，请参阅《State Management in Apache Flink》（图 10-32），论文详细讨论了相关的实现。

State Management in Apache Flink®

Consistent Stateful Distributed Stream Processing

Paris Carbone[†]
Seif Haridi[†]

Stephan Ewen[‡]
Stefan Richter[‡]

Gyula Fóra^{*}
Kostas Tzoumas[‡]

[†]KTH Royal Institute of Technology
{parisc,haridi}@kth.se

^{*}King Digital Entertainment Limited
gyula.fora@king.com

[‡]data Artisans
{stephan,s.richter,kostas}
@data-artisans.com

ABSTRACT

Stream processors are emerging in industry as an apparatus that drives analytical but also mission critical services handling the core of persistent application logic. Thus, apart from scalability and low-latency, a rising system need is first-class support for application state together with strong consistency guarantees, and adaptivity to cluster reconfigurations, software patches and partial failures. Although prior systems research has addressed some of these specific problems, the practical challenge lies on how such guarantees can be materialized in a transparent, non-intrusive manner that relieves the user from unnecessary constraints. Such needs served as the main design principles of state management in Apache Flink, an open source, scalable stream processor.

We present Flink's core pipelined, in-flight mechanism which guarantees the creation of lightweight, consistent, distributed snapshots of application state, progressively, without impacting continuous execution. Consistent snapshots cover all needs for system reconfiguration, fault tolerance and version management through coarse grained rollback recovery. Application state is declared explicitly to the system, allowing efficient partitioning and transparent commits to persistent storage. We further present Flink's backend implementations and mechanisms for high availability, external state queries and output commit. Finally, we demonstrate how these mechanisms behave in practice with metrics and large-deployment insights exhibiting the low performance trade-offs of our approach and the general benefits of exploiting asynchrony in continuous, yet sustainable system deployments.

as a paradigm to implement both analytical applications on “real-time” data, but also as a paradigm to implement data-driven applications and services that would otherwise interact with a shared external database for their data access needs. The stream processing paradigm is more friendly to modern organizations that separate engineering teams vertically, each team being responsible for a specific feature or application, as it allows state to be distributed and co-located with the application instead of forcing teams to collaborate by sharing access to the database. Further, stream processing is a natural paradigm for *event-driven* applications that need to react fast to real-world events and communicate with each other via message passing.

In point of fact, stream processing is not a new concept; it has been an active research topic for the database community in the past [29, 26, 17, 21] and some (but not all) of the ideas that underpin modern stream processing technology are inspired by that research. However, what we see today is widespread adoption of stream processing across the enterprise beyond niche applications where stream processing and Complex Event Processing systems were traditionally used. There are many reasons for this: first, new stream processing technologies allow for massive scale-out, similar to MapReduce [31] and related technologies [46, 20, 22]. Second, the amount of data that is generated in the form of event streams is exploding. Processing needs now spread beyond financial transactions, to user activity in websites and mobile apps, as well as data generated by machines and sensors in manufacturing plants, cars, home devices, etc. Third, many modern state of the art stream processing systems are open source allowing widespread adoption in

图 10-32 《State Management in Apache Flink》

除了保存点之外，Flink 社区还在不断创新，包括将第一个实用流式 SQL API 推向大规模分布式流处理引擎的领域，正如我们在第 8 章中所讨论的那样。总之，Flink 的迅速崛起成为流计算领军角色主要归功于三个特点：

1. 整合行业里面现有的最佳想法（例如，成为第一个开源 DataFlow/Beam 模型）
2. 创新性在表上做了大量优化，并将状态管理发挥更大价值，例如基于 Snapshot 的强一致性语义保证，Savepoints 以及流式 SQL。
3. 迅速且持续地推动上述需求落地。

另外，所有这些改进都是在开源社区中完成的，我们可以看到为什么 Flink 一直在不断提高整个行业的流计算处理标准。

Beam

我们今天谈到的最后一个系统是 Apache Beam (图 10-33)。 Beam 与本章中的大多数其他系统的不同之处在于，它主要是编程模型，API 设计和可移植层，而不是带有执行引擎的完整系统栈。但这正是我想强调的重点：正如 SQL 作为声明性数据处理的通用语言一样，Beam 的目标是成为程序化数据处理的通用语言。

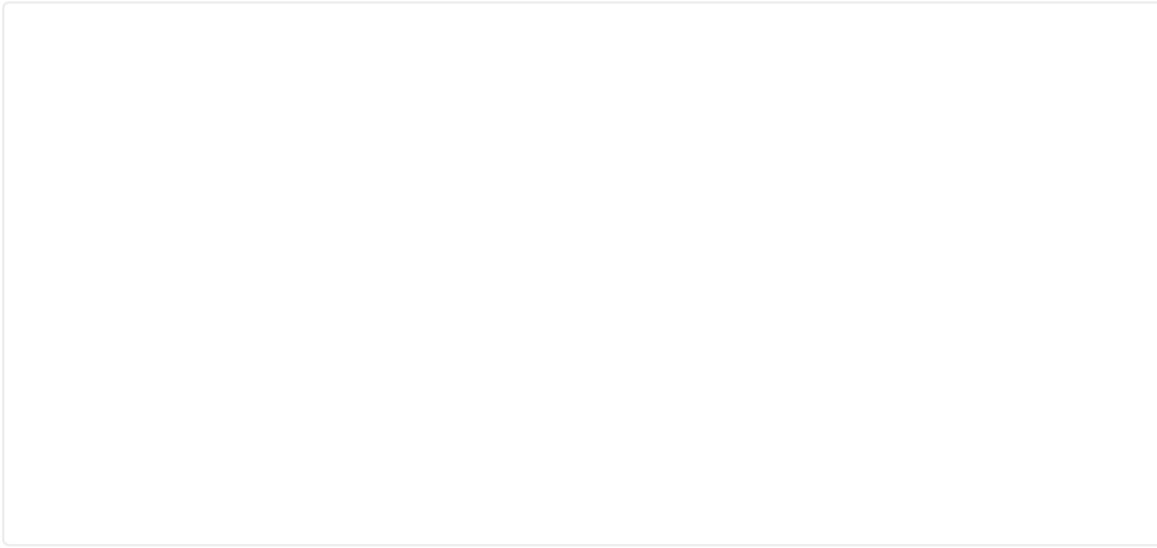


图 10-33 Apache Beam 的时间轴

具体而言，Beam 由许多组件组成：

- 一个统一的批量加流式编程模型，继承自 Google DataFlow 产品设计，以及我们在本书的大部分内容中讨论的细节。该模型独立于任何语言实现或 runtime 系统。您可以将此视为 Beam 等同于描述关系代数模型的 SQL。
- 一组实现该模型的 SDK（软件开发工具包），允许底层的 Pipeline 以不同 API 语言的惯用方式编排数据处理模型。Beam 目前提供 Java，Python 和 Go 的 SDK，可以将它们视为 Beam 的 SQL 语言本身的程序化等价物。
- 一组基于 SDK 的 DSL（特定于域的语言），提供专门的接口，以独特的方式描述模型在不同领域的接口设计。SDK 来描述上述模型处理能力的全集，但 DSL 描述一些特定领域的处理逻辑。Beam 目前提供了一个名为 Scio 的 Scala DSL 和一个 SQL DSL，它们都位于现有 Java SDK 之上。

- 一组可以执行 Beam Pipeline 的执行引擎。执行引擎采用 Beam SDK 术语中描述的逻辑 Pipeline，并尽可能高效地将它们转换为可以执行的物理计划。目前，针对 Apex , Flink , Spark 和 Google Cloud Dataflow 存在对应的 Beam 引擎适配。在 SQL 术语中，您可以将这些引擎适配视为 Beam 在各种 SQL 数据库的实现，例如 Postgres , MySQL , Oracle 等。

Beam 的核心愿景是实现一套可移植接口层，最引人注目的功能之一是它计划支持完整的跨语言可移植性。尽管最终目标尚未完全完成（但即将面市），让 Beam 在 SDK 和引擎适配之间提供足够高效的抽象层，从而实现 SDK 和引擎适配之间的任意切换。我们畅想的是，用 JavaScript SDK 编写的数据 Pipeline 可以在用 Haskell 编写的引擎适配层上无缝地执行，即使 Haskell 编写的引擎适配本身没有执行 JavaScript 代码的能力。

作为一个抽象层，Beam 如何定位自己和底层引擎关系，对于确保 Beam 实际为社区带来价值至关重要，我们也不希望看到 Beam 引入一个不必要的抽象层。这里的关键点是，Beam 的目标永远不仅仅是其所有底层引擎功能的交集（类似最小公分母）或超集（类似厨房水槽）。相反，它旨在为整个社区大数据计算引擎提供最佳的想法指导。这里面有两个创新的角度：

- **Beam 本身的创新**

Beam 将会提出一些 API，这些 API 需要底层 runtime 改造支持，并非所有底层引擎最初都支持这些功能。这没关系，随着时间的推移，我们希望许多底层引擎将这些功能融入未来版本中；对于那些需要这些功能的业务案例来说，具备这些功能的引擎通常会被业务方选择。

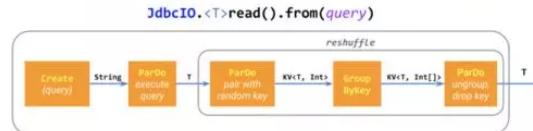
Powerful and modular IO connectors with Splittable DoFn in Apache Beam

Aug 16, 2017 • Eugene Kirpichov

One of the most important parts of the Apache Beam ecosystem is its quickly growing set of connectors that allow Beam pipelines to read and write data to various data storage systems ("IOs"). Currently, Beam ships over 20 IO connectors with many more in active development. As user demands for IO connectors grew, our work on improving the related Beam APIs (in particular, the Source API) produced an unexpected result: a generalization of Beam's most basic primitive, `DoFn`.

Connectors as mini-pipelines

One of the main reasons for this vibrant IO connector ecosystem is that developing a basic IO is relatively straightforward: many connector implementations are simply mini-pipelines (composite `PTransforms`) made of the basic Beam `ParDo` and `GroupByKey` primitives. For example, `ElasticsearchIO.write()` expands into a single `ParDo` with some batching for performance; `JdbcIO.read()` expands into `Create.of(query)`, a reshuffle to prevent fusion, and `ParDo(execute sub-query)`. Some IOs construct considerably more complicated pipelines.



This "mini-pipeline" approach is flexible, modular, and generalizes to data sources that read from a dynamically computed `PCollection` of locations, such as `SpannerIO.readAll()` which reads the results of a `PCollection` of queries from Cloud Spanner, compared to `SpannerIO.read()` which executes a single query. We believe such dynamic data sources are a very useful capability, often overlooked by other data processing frameworks.

When ParDo and GroupByKey are not enough

Despite the flexibility of `ParDo`, `GroupByKey` and their derivatives, in some cases building an efficient IO connector requires extra capabilities.

For example, imagine reading files using the sequence `ParDo(filepattern -> expand into files)`, `ParDo(filename -> read records)`, or reading a Kafka topic using `ParDo(topic -> list partitions)`, `ParDo(topic, partition -> read records)`. This approach has two big issues:

- In the file example, some files might be much larger than others, so the second `ParDo` may have very long individual `@ProcessElement` calls. As a result, the pipeline can suffer from poor performance due to stragglers.
- In the Kafka example, implementing the second `ParDo` is *simply impossible* with a regular `DoFn`, because it would need to output an infinite number of records per each input element `topic, partition` (*stateful processing comes close, but it has other limitations that make it insufficient for this task*).

Beam Source API

Apache Beam historically provides a Source API (`BoundedSource` and `UnboundedSource`) which does not have these limitations and allows development of efficient data sources for batch and streaming systems. Pipelines use this API via the `Read.fromSource` built-in `PTransform`.

The Source API is largely similar to that of most other data processing frameworks, and allows the system to read data in parallel using multiple workers, as well as checkpoint and resume reading from an unbounded data source. Additionally, the Beam `BoundedSource` API provides advanced features such as progress reporting and *dynamic rebalancing* (which together enable autoscaling), and `UnboundedSource` supports reporting the source's watermark and backlog (*until SDF we believed that "batch" and "streaming" data sources are fundamentally different and thus require*

图 10-34 《Powerful and modular I/O connectors with Splittable DoFn in Apache Beam》

这里举一个 Beam 里面关于 `SplittableDoFn` 的 API 例子，这个 API 可以用来实现一个可组合的，可扩展的数据源。（具体参看 Eugene Kirpichov 在他的文章《“Powerful and modular I/O connectors with Splittable DoFn in Apache Beam》中描述 [图 10-34]）。它设计确实很有特点且功能强大，目前我们还没有看到所有底层引擎对动态负载均衡等一些更具创新性功能进行广泛支持。然而，我们预计这些功能将随着时间的推移而持续加入底层引擎支持的范围。

• 底层引擎的创新

底层引擎适配可能会引入底层引擎所独特的功能，而 Beam 最初可能并未提供 API 支持。这没关系，随着时间的推移，已证明其有用性的引擎功能将在 Beam API 逐步实现。

这里的一个例子是 Flink 中的状态快照机制，或者我们之前讨论过的 Savepoints。Flink 仍然是唯一一个以这种方式支持快照的公开流处理系统，但是 Beam 提出了一个围绕快照的 API 建议，因为我们相信数据 Pipeline 运行时优雅更新对于整个行业都至关重要。如果我们今天推出这样的 API，Flink 将是唯一支持它的底层引擎系统。但同样没关系，这里的重点是随着时间的推移，整个行业将开始迎头赶上，因为这些功能的价值会逐步为人所知。这些变化对每个人来说都是一件好事。

通过鼓励 Beam 本身以及引擎的创新，我们希望推进整个行业快速演化，而不用再接受功能妥协。通过实现跨执行引擎的可移植性承诺，我们希望将 Beam 建立为表达程序化数据处理流水线的通用语言，类似于当今 SQL 作为声明性数据处理的通用处理方式。这是一个雄心勃勃的目标，我们并没有完全实现这个计划，到目前为止我们还有很长的路要走。

总 结

我们对数据处理技术的十五年发展进行了蜻蜓点水般的回顾，重点关注那些推动流式计算发展的关键系统和关键思想。来，最后，我们再做一次总结：

- **MapReduce：可扩展性和简单性** 通过在强大且可扩展的执行引擎之上提供一组简单地数据处理抽象，MapReduce 让我们的数据工程师专注于他们的数据处理需求的业务逻辑，而不是去构建能够适应

在一大堆普通商用服务器上的大规模分布式处理程序。

- **Hadoop : 开源生态系统** 通过构建一个关于 MapReduce 的开源平台，无意中创建了一个蓬勃发展的生态系统，其影响力所及的范围远远超出了其最初 Hadoop 的范围，每年有大量的创新性想法在 Hadoop 社区蓬勃发展。
- **Flume : 管道及优化** 通过将逻辑流水线操作的高级概念与智能优化器相结合，Flume 可以编写简洁且可维护的 Pipeline，其功能突破了 MapReduce 的 Map→Shuffle→Reduce 的限制，而不会牺牲性能。
- **Storm : 弱一致性，低延迟** 通过牺牲结果的正确性以减少延迟，Storm 为大众带来了流计算，并开创了 Lambda 架构的时代，其中弱一致的流处理引擎与强大一致的批处理系统一起运行，以实现真正的业务目标低延迟，最终一致型的结果。
- **Spark: 强一致性** 通过利用强大一致的批处理引擎的重复运行来提供无界数据集的连续处理，Spark Streaming 证明至少对于有序数据集的情况，可以同时具有正确性和低延迟结果。
- **MillWheel : 乱序处理** 通过将强一致性、精确一次处理与用于推测时间的工具（如水印和定时器）相结合，MillWheel 做到了无序数据进行准确的流式处理。
- **Kafka: 持久化的流式存储，流和表对偶性** 通过将持久化数据日志的概念应用于流传输问题，Kafka 支持了流式数据可重放功能。通过对流和表理论的概念进行推广，阐明数据处理的概念基础。
- **Cloud Dataflow : 统一批流处理引擎** 通过将 MillWheel 的无序流式处理与高阶抽象、自动优化的 Flume 相结合，Cloud Dataflow 为批流数据处理提供了统一模型，并且灵活地平衡正确性、计算延迟、成本的关系。

- **Flink：开源流处理创新者** 通过快速将无序流式数据处理的强大功能带到开源世界，并将其与分布式快照及保存点功能等自身创新相结合，Flink 提高了开源流处理的业界标准并引领了当前流式处理创新趋势。
- **Beam: 可移植性** 通过提供整合行业最佳创意的强大抽象层，Beam 提供了一个可移植 API 抽象，其定位为与 SQL 提供的声明性通用语言等效的程序接口，同时也鼓励在整个行业中推进创新。

可以肯定的说，我在这里强调的这 10 个项目及其成就的说明并没有超出当前大数据的历史发展。但是，它们对我来说是一系列重要且值得注意的大数据发展里程碑，它共同描绘了过去十五年中流处理演变的时间轴。自最早的 MapReduce 系统开始，尽管沿途有许多起伏波折，但不知不觉我们已经走出来很长一段征程。即便如此，在流式系统领域，未来我们仍然面临着一系列的问题亟待解决。正所谓：路漫漫其修远兮，吾将上下而求索。

译者简介

陈守元（花名：巴真），阿里巴巴高级产品专家。阿里巴巴实时计算团队产品负责人，2010 年毕业即加入阿里集团参与淘宝数据平台建设，近 10 年的大数据从业经验，开源项目 Alibaba DataX 发起人，当前负责阿里实时计算产品 Flink 的规划与设计，致力于推动 Flink 成为下一代大数据处理标准。

《Streaming System》一书目前正由阿里巴巴实时计算团队进行翻译，预计今年年底上市，对流式系统感兴趣的的同学可以关注。

今日荐文

点击下方图片即可阅读

[开源Kubeflow：在Kubernetes上运行机器学习](#)



会议推荐

近几年，短视频应用蓬勃发展，由于短视频场景下用户兴趣和广告内容更难以理解，短视频广告在用户内容理解、召回、排序和机制上都会遇到更大挑战。基于快手海量的用户和视频数据，利用 AI 相关技术，可以更好的解决这些问题。

AICon 全球人工智能与机器学习技术大会上，快手短视频商业化模型方向负责人孔东营将会为我们带来“机器学习在短视频商业化中如何应用”相关议题的精彩分享。

除了机器学习，更多计算机视觉、NLP、自动驾驶、知识图谱、搜索推荐与算法、AI 工具与框架等热门议题欢迎扫描下方二维码或点击“阅读原文”了解详情。目前大会报名八折优惠中，详情咨询：18514549229（同微信）。

如果你喜欢这篇文章，或希望看到更多类似优质报道，记得给我留言和点赞哦！

[Read more](#)