大规模数据处理实战——必要性与发展趋势

参考自己未来规划以及找工作的进展，以后主要会从事大数据开发的相关工作，所以学习了极客时间上的大规模数据处理实战这门课程。后续会分几次总结课程内容。本文主要介绍大规模数据处理的必要性以及大规模数据处理技术的发展趋势，并结合自己在面试过程中的经验，总结一些面试常见问题。

大规模数据处理的必要性

在面试时，有些面试官会说请介绍一下你理解的大数据技术，这个时候我通常都会从大数据技术产生的必要性谈起。

数据处理的重要性

没有高质量的数据处理的话，大火的人工智能就只有人工没有智能。

我们不知道的是，Google 曾在很长的一段时间里低估过数据处理。例如，在语义理解上，Google 认为自己有最多的搜索文本数据，最好的算法，那就一定能把语义理解做的最好。可是到 2016 年左右，一个名不见经传的德国小公司却一举超过了 Google，大家都很惊讶，后来发现原来他们凭借的是高质量的数据标注和处理。

所以，高质量的数据处理，能够为后续的知识挖掘、模型训练等很多任务提供重要的价值，这也是很多时候数据处理的意义。

数据处理工程师在组织架构上的重要性

许多工程师都喜欢自嘲自己的工作是“搬砖”，事实也正是如此。很多人的工作内容都避不开数据的搬运和处理。把数据从这个格式处理成那个格式，把数据从这个数据库搬到那个数据库，这个服务器搬到那个服务器，这个客户端搬到那个客户端。可能大家都还没有意识到，即使是一个写前端的工程师，他的很多工作还是数据处理。大数据领域泰斗级人物 Jesse Anderson 曾做过一项研究，一个人工智能团队的合理组织架构，需要五分之四的数据处理工程师。很不幸，很多团队没有认识到这一点。

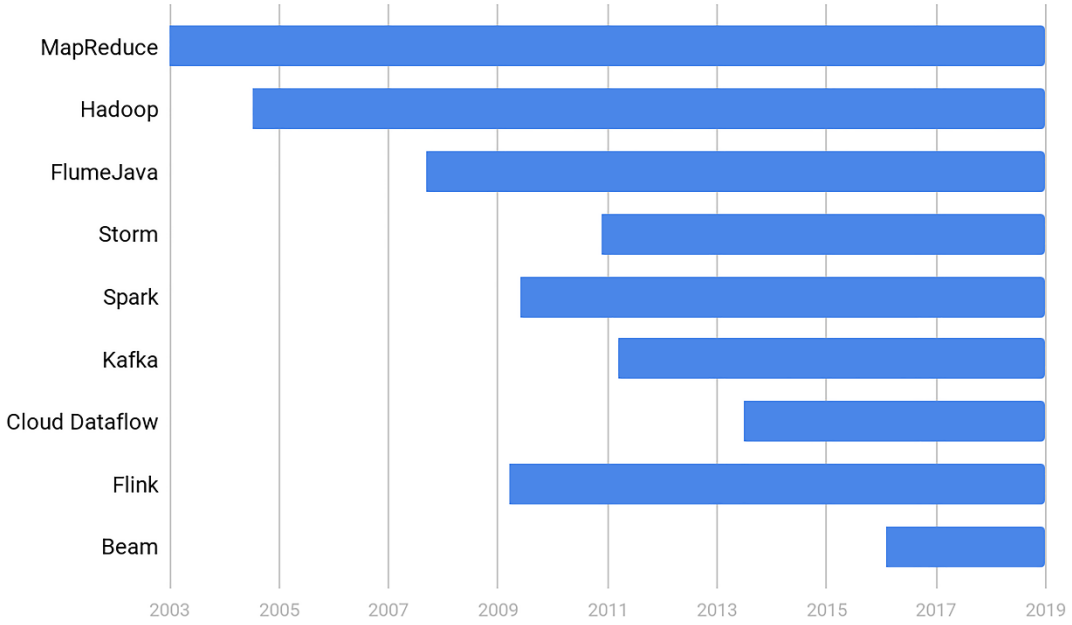
所以，在目前的大部分团队或者项目中，数据处理工程师都是十分重要的而有价值的。

数据处理规模变大带来的复杂度

面试过程中，经常会有很编程问题，但是面试官会追问如果数据规模变大时，要怎么怎么设计，数据排序等问题。这时很多时候就要借助于大规模数据处理组件进行处理，比如 MapReduce，Spark 等。

大规模数据处理技术的发展趋势

大规模数据处理的重要技术以及它们产生的年代可以用下图概括。



极客时间这门课程的老师将大规模数据处理技术的发展分为三个阶段：石器时代，青铜时代和蒸汽机时代。

石器时代

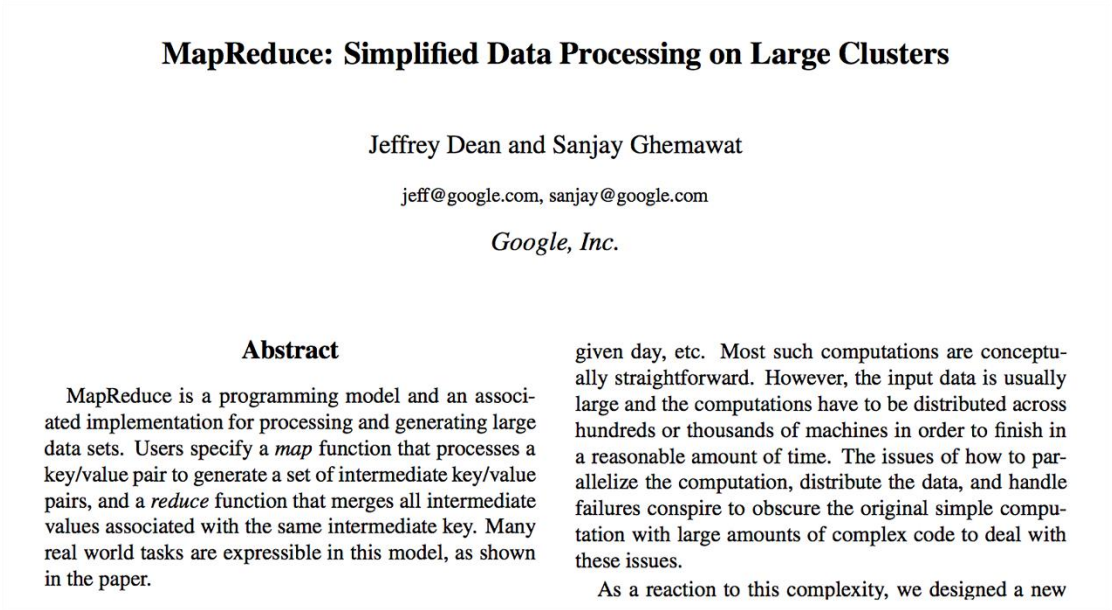
他用“石器时代”来比喻 MapReduce 诞生之前的时期。

数据的大规模处理问题早已存在。早在 2003 年的时候，Google 就已经面对大于 600 亿的搜索量。但是当时数据的大规模处理技术还处在彷徨阶段。当时每个公司或者个人可能都有自己的一套工具处理数据。却没有提炼抽象出一个系统的方法。

如果面对大规模数据处理问题的每个人都自己开发一个系统，或自己想一种方法来解决这一问题的话，无疑这是十分困难且浪费人力物力的，所以大规模数据处理技术应运而生。

青铜时代

2003 年，MapReduce 的诞生标志了超大规模数据处理的第一次革命，而开创这段青铜时代的就是下面这篇论文《MapReduce: Simplified Data Processing on Large Clusters》。



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.

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

Jeff Dean 和 Sanjay Ghemawat 从纷繁复杂的业务逻辑中，为我们抽象出了 Map 和 Reduce 这样足够通用的编程模型。后面的 Hadoop 仅仅是对于 GFS、BigTable、MapReduce 的依葫芦画瓢与集成。

蒸汽机时代

到了 2014 年左右，Google 内部已经几乎没人写新的 MapReduce 了。2016 年开始，Google 在新员工的培训中把 MapReduce 替换成了内部称为 FlumeJava（不是 Apache Flume，是不同的两个技术）的数据处理技术。这标志着青铜时代的终结，同时也标志着蒸汽机时代的开始。之所以跳过“铁器时代”之类的描述，是因为只有工业革命的概念才能解释从 MapReduce 进化到 FlumeJava 的划时代意义。

2010 年以来，又陆续出现了 spark、storm、kafka 等大数据处理技术组件，更加丰富了数据处理工程师的工具库，使得数据处理工程师在面对不同的业务场景时，都能够有适合这个场景的开源工具可以使用。

面试常见问题

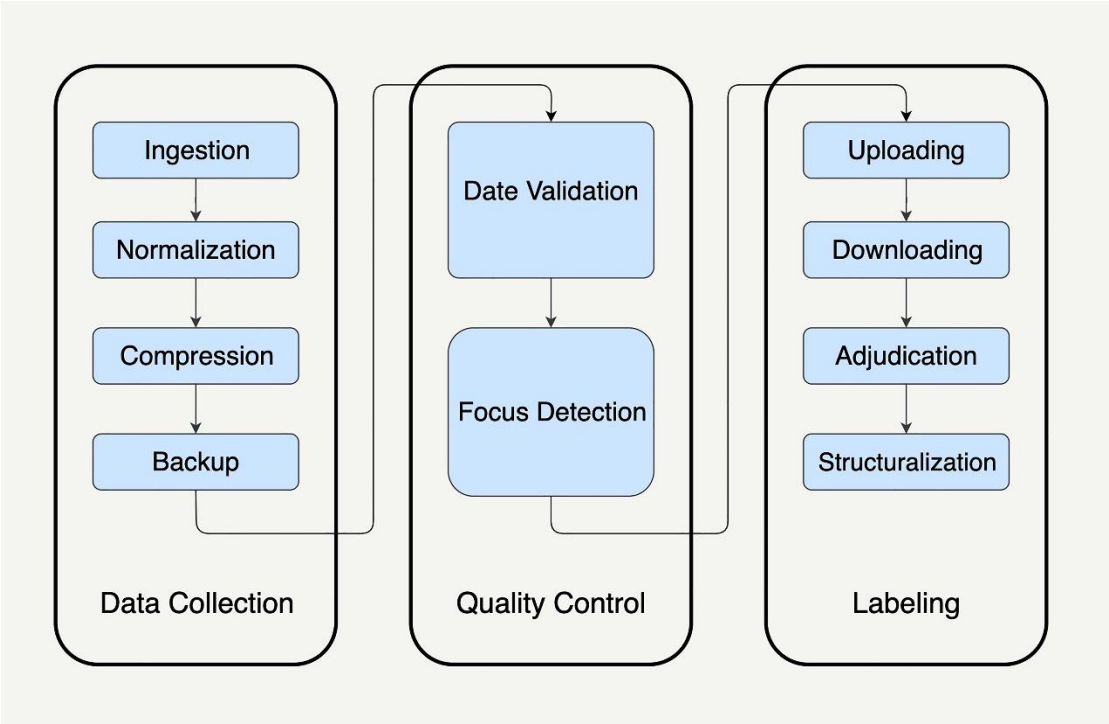
为什么大家都不用 MapReduce 了？

高昂的维护成本

使用 MapReduce，你需要严格地遵循分步的 Map 和 Reduce 步骤。当你构造更为复杂的处理架构时，往往需要协调多个 Map 和多个 Reduce 任务。然而，每一步的 MapReduce 都有可能出错。为了这些异常处理，很多人开始设计自己的协调系统（orchestration）。例如，做一个状态机（state machine）协调多个 MapReduce，这大大增加了整个系统的复杂度。如果你搜“MapReduce orchestration”这样的关键词，就会发现有很多书，整整一本都在写怎样协调 MapReduce。你可能会惊讶于 MapReduce 的复杂度。

有时也会看到一些把 MapReduce 说得过度简单的误导性文章。例如，“把海量的××数据通过 MapReduce 导入大数据系统学习，就能产生××人工智能”。似乎写文的“专家”动动嘴就能点石成金。而现实的 MapReduce 系统的复杂度是超过了“伪专家”的认知范围的。

举个例子，想象一下这个情景，你的公司要预测美团的股价，其中一个重要特征是活跃在街头的美团外卖电动车数量，而你负责处理所有美团外卖电动车的图片。在真实的商用环境下，为了解决这个问题，你可能至少需要 10 个 MapReduce 任务：



首先，我们需要搜集每日的外卖电动车图片。数据的搜集往往不全部是公司独自完成，许多公司会选择部分外包或者众包。所以在数据搜集（Data collection）部分，你至少需要 4 个 MapReduce 任务：

1) 数据导入（data ingestion）：用来把散落的照片（比如众包公司上传到网盘的照片）下载到你的存储系统。
2) 数据统一化（data normalization）：用来把不同外包公司提供过来的各式各样的照片进行格式统一。
3) 数据压缩（compression）：你需要在质量可接受的范围内保持最小的存储资源消耗 。
4) 数据备份（backup）：大规模的数据处理系统我们都需要一定的数据冗余来降低风险。

仅仅是做完数据搜集这一步，离真正的业务应用还差得远。

真实的世界是如此不完美，我们还需要一部分数据质量控制（quality control）流程，比如：

1) 数据时间有效性验证 （date validation）：检测上传的图片是否是你想要的日期的。
2) 照片对焦检测（focus detection）：你需要筛选掉那些因对焦不准而无法使用的照片。

最后才到你负责的重头戏——找到这些图片里的外卖电动车。而这一步因为人工的介入是最难控制时间的。你需要做 4 步：

1) 数据标注问题上传（question uploading）：上传你的标注工具，让你的标注者开始工作。
2) 标注结果下载（answer downloading）：抓取标注完的数据。

3) 标注异议整合（adjudication）：标注异议经常发生，比如一个标注者认为是美团外卖电动车，另一个标注者认为是京东快递电动车。
4) 标注结果结构化（structuralization）：要让标注结果可用，你需要把可能非结构化的标注结果转化成你的存储系统接受的结构。

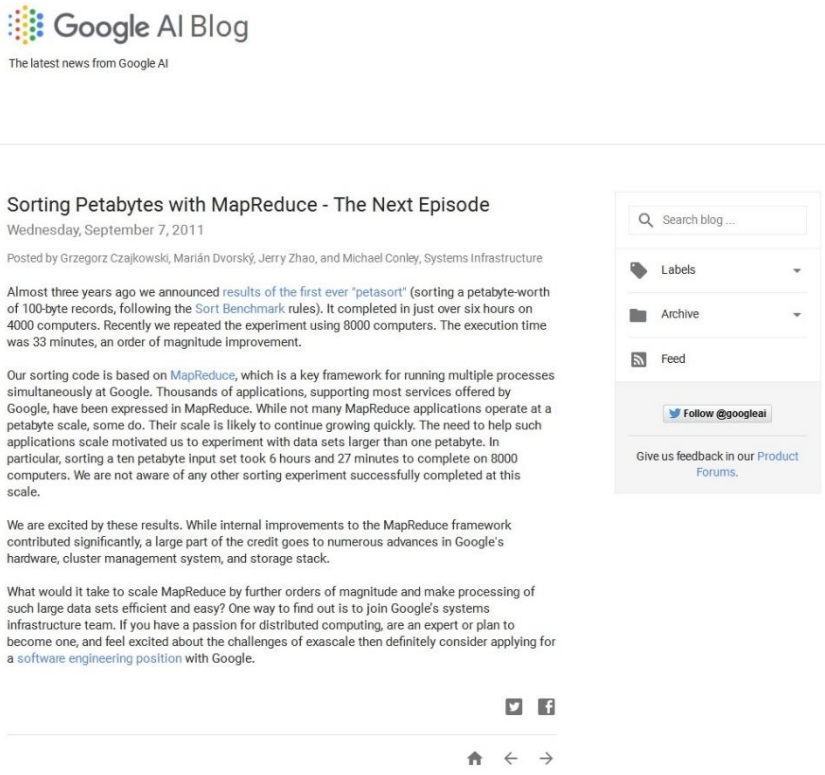
同上上述简单分析足可见使用 MapReduce 在面对较复杂场景是有多力不从心。

在应用过程中，每一个 MapReduce 任务都有可能出错，都需要重试和异常处理的机制。所以，协调这些子 MapReduce 的任务往往需要和业务逻辑紧密耦合的状态机。这样过于复杂的维护让系统开发者苦不堪言。

时间性能“达不到”用户的期待

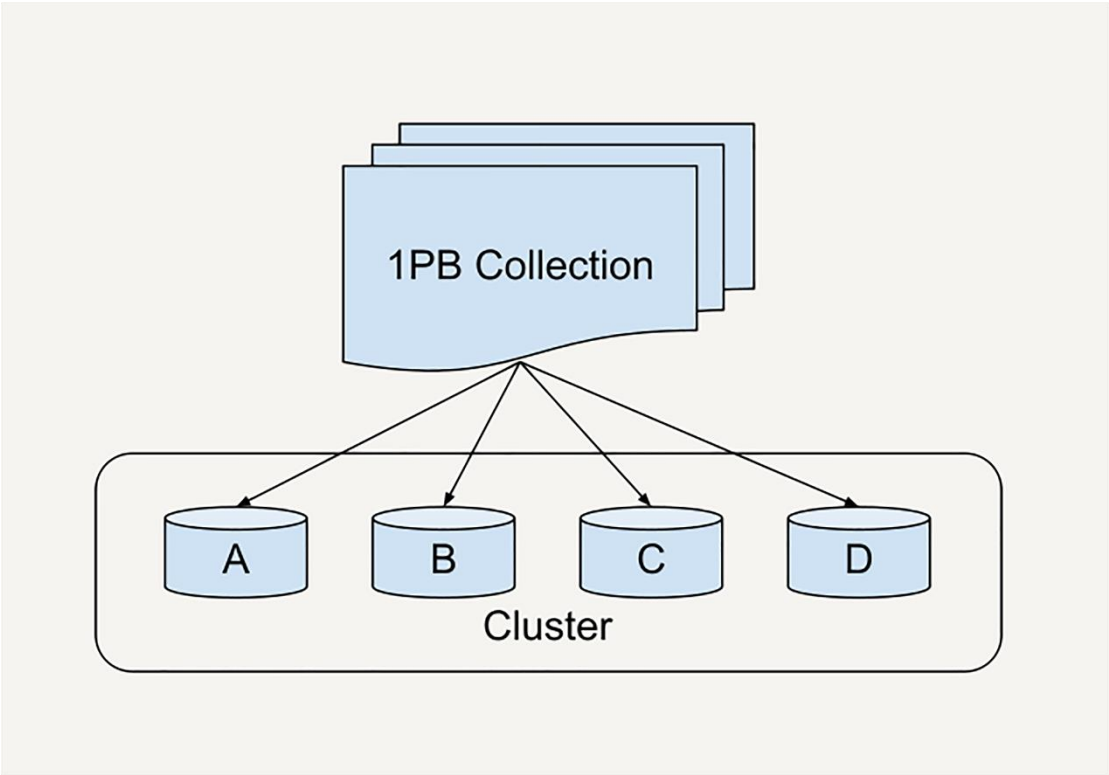
除了高昂的维护成本，MapReduce 的时间性能也是个棘手的问题。MapReduce 是一套如此精巧复杂的系统，如果使用得当，它是青龙偃月刀，如果使用不当，它就是一堆废铁。不幸的是并不是每个人都是关羽。在实际的工作中，不是每个人都对 MapReduce 细微的配置细节了如指掌。在现实中，业务往往需求一个刚毕业的新手在 3 个月内上线一套数据处理系统，而他很可能从来没有用过 MapReduce。这种情况下开发的系统是很难发挥好 MapReduce 的性能的。

在 Google 内部，MapReduce 的性能优化手册足足有 500 多页。以 MapReduce 的分片（sharding）难题举例。Google 曾经在 2007 年到 2012 年间做过一个对于 1PB 数据的大规模排序实验，来测试 MapReduce 的性能。从 2007 年的排序时间 12 小时，到 2012 年的排序时间缩短至 0.5 小时。即使是 Google，也花了 5 年的时间才不断优化了一个 MapReduce 流程的效率。2011 年，他们在 Google Research 的博客上公布了初步的成果。

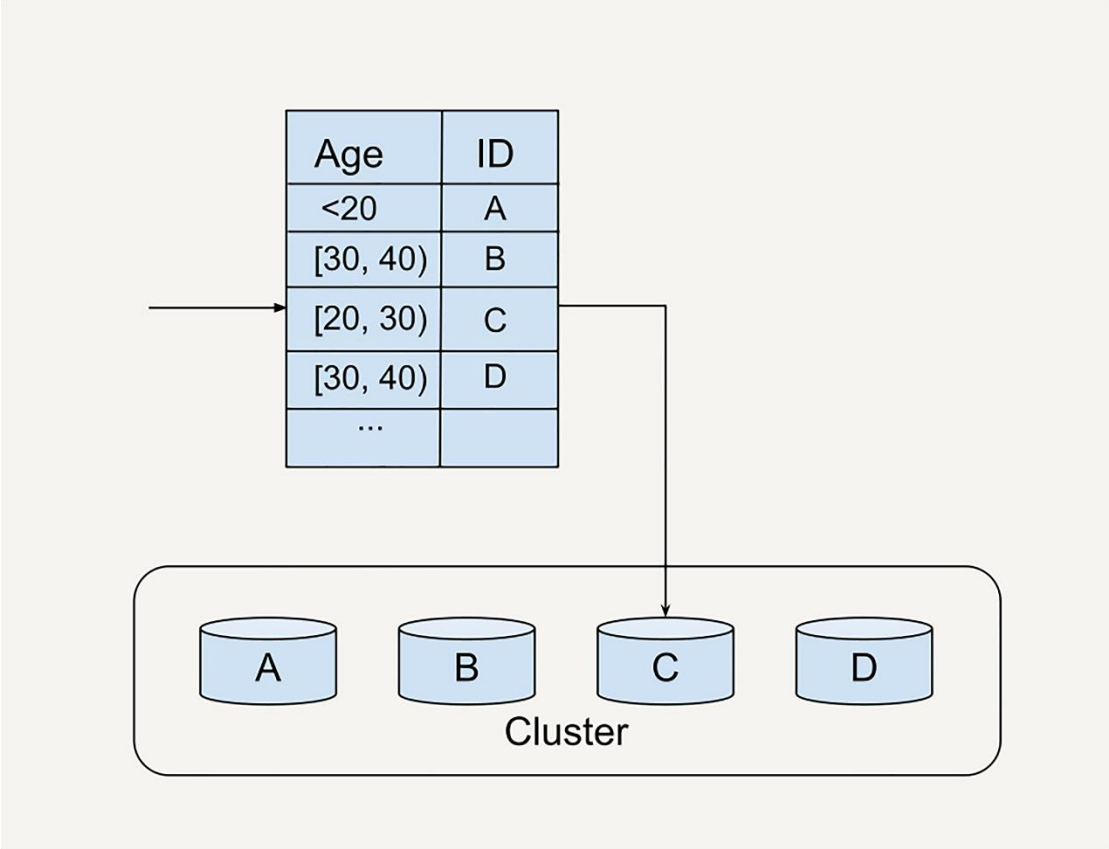


其中有一个重要的发现，就是他们在 MapReduce 的性能配置上花了非常多的时间。包括了缓冲大小（buffer size），分片多少（number of shards），预抓取策略（prefetch），缓存大小（cache size）等等。

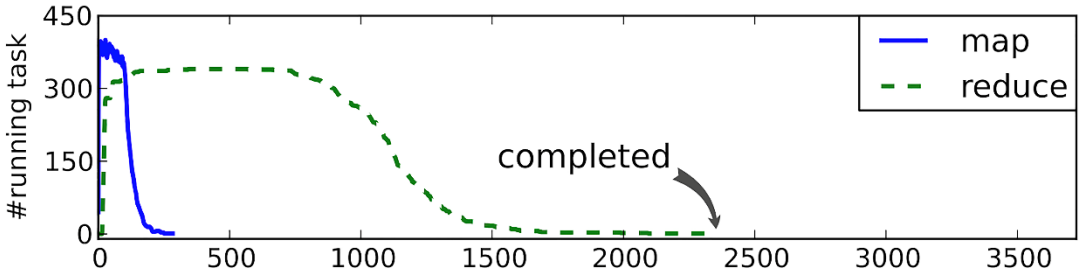
所谓的分片，是指把大规模的的数据分配给不同的机器/工人，流程如下图所示。



选择一个好的分片函数（sharding function）为何格外重要？让我们来看一个例子。假如你在处理 Facebook 的所有用户数据，你选择了按照用户的年龄作为分片函数（sharding function）。我们来看看这时候会发生什么。因为用户的年龄分布不均衡（假如在 20~30 这个年龄段的 Facebook 用户最多），导致我们在下图中 worker C 上分配到的任务远大于别的机器上的任务量。



这时候就会发生掉队者问题（stragglers）。别的机器都完成了 Reduce 阶段，只有 worker C 还在工作。当然它也有改进方法。掉队者问题可以通过 MapReduce 的性能剖析（profiling）发现。 如下图所示，箭头处就是掉队的机器。



回到刚刚的 Google 大规模排序实验。因为 MapReduce 的分片配置异常复杂，在 2008 年以后，Google 改进了 MapReduce 的分片功能，引进了动态分片技术（dynamic sharding），大大简化了使用者对于分片的手工调整。在这之后，包括动态分片技术在内的各种崭新思想被逐渐引进，奠定了下一代大规模数据处理技术的雏型。

MapReduce 和 Spark 的区别（这个问题我被问过好几次）

结合上面讲到的 MapReduce 的特点，可以总结 spark 和 MapReduce 区别如下：

1) 相比 MapReduce 面对复杂场景需要多轮 map 和 reduce 任务，spark 的处理逻辑是连贯的，可以通过一个 spark 任务内部的多个算子来完成复杂的逻辑，执行流程是 DAG（一个有向无环图），不需要额外系统进行协调；
2) Spark 的速度一般比 MapReduce 快。MapReduce 作为 hadoop 的组件，每一轮 map

和 reduce 任务完成后，结果都是要落地写入 hdfs 的，大大增多了磁盘 IO 时间，而spark是基于内存进行计算的，上一步的结果是存在内存中供下一步计算使用的，速度明显要快。

3) Spark 容错性高。MapReduce 需要重新计算，但是 spark 可以利用内部的 RDD，根据计算时的 DAG 进行恢复。
4) Spark 更加通用，组件包括 spark SQL，streaming，GraphX 等，另外还有很多算子操作；而 MapReduce 只有 map 和 reduce 任务。
5) Spark 和 MapReduce 都有 shuffle 步骤，但是 MapReduce 的 shuffle 是基于排序的，而 spark 是基于 hash 的。