Google 系列论文翻译集

译者：phylips@bmy 2011-7

本系列包含如下几篇论文：

Cluster：发表于 2003 年，主要介绍 Google 的集群架构，对 Google 搜索系统的架构也进行了简单介绍

GFS：发表于 2003 年，介绍了 Google 分布式文件系统的设计及实现。Hadoop 中与之对应的是 HDFS

MapReduce：发表于 2004 年，介绍了分布式的编程模型 MapReduce。Hadoop 中与之对应的是 Hadoop MapReduce

BigTable：发表于 2006，介绍了建立在 GFS 之上的结构化数据存储系统，该系统也是 NoSql 的。Hadoop 中与之对应的是 HBase

Chubby：发表于 2006 年，分布式锁服务系统，利用了很多现有的思想，尤其是分布式系统中的很多基础理论。Hadoop 中与之对应的是 Zookeeper

Sawzall:发表于 2006 年,建立在 MapReduce 之上的分布式查询脚本语言。Hadoop 中与之对应的是 Pig Hive 等

GFS 访谈录：Kirk McKusick 与 Sean Quinlan 之间关于 GFS 的起源和演化的访谈内容。可能比 GFS 本身更有意思。

SMAQ：关于海量数据的存储计算及查询的一个综述性文章。将视野从 Google 系列扩展到了当今流行的各种 NoSql 系统。

序

大概一年前，翻译了 google 的 Cluster,GFS,MapReduce,BigTable 这几篇论文。随着工作的进行及参阅资料的增加，是时候重新看下当年翻译的内容了。看后也发现了一些笔误及理解不到位的地方，于是应该重新做下修正。但是由于之前的文章散落在 blog 上的各处，而篇幅又都比较长，实在懒得修改原来的内容。幸好我通常都会将 blog 上的文章先在本地编辑，所以干脆直接修改本地的版本吧，然后整理在一块。

所以呢，一方面重新检查下原来翻译的内容，对其中不当的地方予以纠正，同时又添加了一些理解性的文字。另一方面，当初在翻译这些论文时，实际上刻意留下了一篇 Chubby，主要是因为这个涉及到比较基础的分布式理论，当时理解起来显得要比其他几篇吃力，索性就留了下来等以后再翻。现在呢，在看完拜占庭将军问题、FLP 结论、Leases、Paxos 这几个之后，我想重新阅读 Chubby 的时机也到了，所以呢这次就加上了 Chubby 这篇。

此外，如果要构成完整的 SMAQ体系，实际上还应该加上 Sawzall，不过这篇相对容易理解一些，同时网上也已存在比较好的中文版本，所以不再翻译，当然其他几篇网上也基本上都存在中文版本(GFS,MapReduce,BigTable应该很容易找到，关于Cluster和Chubby的则比较难找)，但是它们是如此之重要，为了加深印象和理解，我还是自行将它们翻译了出来，当然这也是当时的初衷了。

当然呢，google 的这些论文都已经是几年前发表的了。之后 google 又发表的一些诸如 Pregel,Dremel,Percolator,MegaStore，则相对小众一些，所以就未纳入。正是 google 发表 GFS,MapReduce,BigTable,Chubby,Sawzall 这些论文之后，才衍生出 Hadoop 这样的一个海量数据处理的生态系统，形成了与之对应的诸如 HDFS,Hadoop MapReduce,Hbase,Zookeeper,Pig,Hive 这样的一些系统。要理解这样的一些系统，也还是要从 google 最原始的论文开始作为起点。

最后呢，其实我们可以看到在这些论文发表之后，一方面 google 本身的系统在不断演化，另一方面，开源实现与原始论文相较具体实现也有所不同，同时当开源软件应用到不同公司之后，这些公司本身会发展出适应自身应用的版本。具体来说：比如 google 的 GFS 在历经 10 多年的发展之后，与原始论文相比已发生了很多变化；再比如 Hadoop 已提出 MapReduce 2，致力于更高的可用性、扩展性及通用性；Facebook 对 Hadoop 针对自身应用特点进行了各种改进，主要是针对其实时性应用的特点，增加可用性及降低延迟。所以在这些论文的最后，又附上了一个 ACM 针对 GFS 的起源及演化，对 google 的 Sean Quinlan 所做的一次访谈。这个访谈已发表在 ACM Queue 上，对于理解一个系统是如何在漫长的时间内不断修正最初的设计、满足各种需求，如何一步步地演化是十分有益的。

由于时间能力及经验的限制，这些内容难免有偏颇及错误之处，欢迎指出其中的不当之处。同时为方便对比英文版阅读，在文章最后附上了对应的英文原版论文。最后，为组成一个完整的海量数据的存储计算和查询模型，又加上了关于 Sawzall(对崮山路上走 9 遍翻译的版本做了一些修正及补充)及 SMAQ 的文章。

面向星球的网络搜索：google 集群架构

作者：Luiz Andre Barroso,Jeffrey Dean,Urs Holzle Google Inc 2003
译者：phylips@bmy 2010-10-1
出处：http://duanple.blog.163.com/blog/static/70971767201091102339246/

为了能够支持可扩展的并行化，google 的网络搜索应用让不同的查询由不同的处理器处理，同时通过划分全局索引，使得单个查询可以利用多个处理器处理。为了能够处理这样的工作负载，google 的集群架构由 15000 个普通 pc 机和容错软件组成。这种架构达到了很高的性能，同时由于采用了普通 pc 机，也节省了采用昂贵的高端服务器所需的大量花费。

在众多的网络服务中，很少有像搜索引擎那样，单个请求占用那样多的计算资源。平均来看，在 google 上的每次查询需要读取数百 m 的数据耗费数 10 亿的 cpu 指令循环。为了能够支持峰值达到每秒数千次的请求量，需要与世界上最大的超级计算机规模相当的硬件设施。通过容错性软件将 15000 个普通 pc 机联合起来，提供了一种比使用高端服务器更廉价的解决方案。

本文我们将介绍 google 的集群架构，讨论那些影响到设计方案的最重要的因素：能效和性价比。在我们的实际操作中，能效实际上是一个关键的度量标准，因为数据中心的电力密度是有限的，因此电力消耗和制冷是一个关键的考虑因素。

我们的应用本身可以很容易进行并行化：不同的查询可以运行在不同的处理器上，同时全局索引也划分的使得单个查询可以使用多个处理器。因此，处理器的性价比比峰值性能变得更重要。同时，google 的应用是面向吞吐率的，可以更有效的利用处理器提供的并行化，比如并行多线程(SMT)，或者多核处理器(CMP)。

Google 架构概览

Google 的软件架构源自两个基本的观点。首先我们需要在软件层面提供可靠性，而不是通过硬件，这样我们就可以使用普通的 pc 构建廉价的高端集群。其次，我们不断地调整设计以达到最好的总体请求吞吐率，而非提高服务器的峰值响应时间， 因为我们可以通过并行化独立的请求来控制响应时间。

我们相信使用不可靠的廉价 pc 来构建可靠的计算设施可以达到最好的性价比。通过在不同的机器上备份服务，以及自动化的故障检测和错误处理，为我们的环境提供软件级的可靠性。这种软件级的可靠性在我们的系统设计中几乎随处可见。审视一下一次查询处理的控制流程，有助于理解这种高级的查询服务系统，同时也有助于理解各种有关可靠性的考虑。

Google 的一次查询

当用户在 google 中输入一次查询，用户浏览器首先通过 DNS 进行域名解析，将 www.google.com 转换为 ip 地址。为了对查询可以进行更有效的处理，我们的服务由分布在世界各地的多个集群组成。每个集群大概有数千个机器，这种地理上的分布可以有效的应付灾难性的数据中心失败比如地震，大规模的停电。基于 DNS 的负载平衡系统，会计算用户与每一个物理集群地理上的距离来选择一个合适的物理集群。负载平衡系统，需要最小化请求往返时间，同时要考虑各个集群的可用容量(capacity)。

然后用户浏览器向其中的一个集群发送一个 http 请求，之后，对于该集群来说，所有的处理都变成了本地的。在每个集群中有一个基于硬件的负载平衡器监控当前可用的 google web servers(GWS)集合，并在这个集合上将本地的请求处理进行负载平衡。收到一个请求之后，GWS 协调这个查询的执行，并将结果格式化为 html 语言。图 1 表示了这个过程。

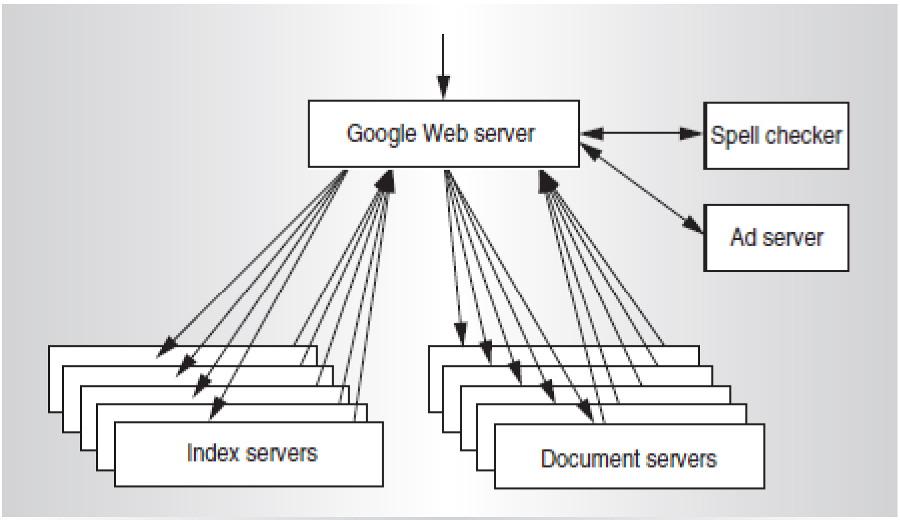


Figure 1. Google query-serving architecture.

查询执行由两个主要阶段组成，第一个阶段，索引服务器查阅倒排索引(将每个查询词映射到匹配的文档列表)。索引服务器然后决定相关的文档集合，通过对每个查询词匹配的文档列表求交集，为每个文档计算出一个相关性的分值，这个分值决定了在输出结果中的排序。

搜索的过程非常具有挑战性，因为需要处理海量数据：原始网页文档通常具有数十 T 的未压缩数据，从原始数据中导出的倒排索引本身也有好几 T 的数据。幸运的是，通过将索引划分到不同的片段，可以将搜索高度并行化，每个片段具有从全部索引中随机选择的一个文档子集。一组机器负责处理对于一个索引片段的请求，在整个集群中每个片段都会有这样的一组机器与之对应。每个请求通过中间负载平衡器选择组内机器中的一个，换句话说每个查询将会访问分配到每个片段的一台机器(或者是一组机器的子集)。如果一个片段的备份坏了，负载平衡器将

会避免在查询时使用它，我们的集群管理系统的其他组件将会尝试修复它，实在不行就用另一台机器来取代它。停机期间，系统的容量需要减去那台坏掉的机器所代表的容量。然而，服务仍然是未中断的的，索引仍然是可用的。

第一阶段的查询执行最终输出一个排过序的文档标识符列表。第二阶段则通过获取这个文档列表，然后计算出所有文档的标题和 url 以及面向查询内容的文档摘要。文档服务器处理这项任务，从硬盘中获取文档，抽取标题以及查询关键词在文档中的出现片段。像索引查找阶段，这里的策略也是对文档进行划分，主要通过：随机分布文档到不同的小片段；针对每个片段的处理具有多个服务器作为备份；通过一个负载平衡器分发请求。文档服务器必须能够访问一个在线的低延迟的全网网页的副本。实际上由于对于这个副本访问的性能及可用性需求，google 实际上会在集群中存储整个 web 的多个副本。

除了索引和文档服务阶段，GWS 在收到查询时还会初始化几个其他的辅助任务，比如将查询发送给拼写检查系统，发给广告系统以生成相关广告。当所有阶段完成后，GWS 生成 html 输出页面，然后返回给用户浏览器。

使用备份进行容量扩充和容错

我们通过对系统进行一些结构上的设计以保证对于索引和其他与查询响应相关的数据结构的访问操作都是只读的：更新是相对不频繁的，这样我们就能通过将查询转移到一个服务备份来安全地进行更新。 这条原则，使得我们避免了很多在通用数据库中出现的一致性问题。

我们也尽力挖掘出大量应用中固有的并行性：比如我们将在一个大索引中的匹配文档的查询转化为针对多个片段中的匹配文档的多个查询加上开销相对便宜的归并步骤。类似的，我们将查询请求流划分为多个流，每个由一个集群来处理。为每个处理索引片段的机器组增加机器来增加系统容量，伴随着索引的增长增长片段的个数。通过将搜索在多个机器上并行化，我们降低了响应一个查询所必需的平均延时，将整个计算任务划分在多个 cpu 和硬盘上。因为独立的片段相互之间不需要通信，所以加速比几乎是线性的。换句话说，单个索引服务器的 cpu 速度不会影响整个搜索的整体性能，因为我们可以增加片段数来适应慢的 cpu，因此我们的硬件选择主要关注那些可以为我们的应用提供出色的请求吞吐率的机器，而不是可以提供最高的单线程性能的那些。

简单来说，google 的集群主要遵循下面三个主要设计原则：

- 软件可靠性。我们没有选择硬件性容错，比如采用冗余电源，RAID，高质量组件，而是专注于软件可靠性。
- 使用备份得到更好的吞吐率和可用性。因为机器本身是不可靠的，我们将我们的内部服务备份在很多机器上。通过备份我们得到了容量(Capicity,这样不同的备份可以同时提供读取服务，因此提高了能够服务的客户端数)，与此同时也得到了容错，而这种容错几乎是免费的。

- 性价比重于峰值性能。我们购买当前最具性价比的 cpu，而不是那些具有最高绝对性能的 cpu。
- 使用普通 pc 降低计算花费。这样我们就可以为每一个查询提供更多的计算资源，可以在 ranking 算法中使用开销更高的技术，可以搜索更大的文档索引。

使用商品化部件

Google 的机柜是专门定制的，由两面组成，总共放置了 40 到 80 个基于 80x86 的服务器(每侧包含 20 个 20u 或者 40 个 10u 服务器) 。我们对于性价比的偏爱，使得我们直接选择中端桌面 pc 的组件进行组装，但是会选择大一些的硬盘驱动器。目前的服务中使用了好几个 cpu 产品，从 533m intel-celeron 到双核 1.4G Intel 奔三服务器。每个服务器包含一个或者多个 IDE 硬盘，每个 80g 空间。与文档服务器相比，索引服务器通常具有更少的磁盘空间，因为它的负载类型是对 cpu 更敏感。在机柜一侧的服务器通过一个 100-Mbps 以太网交换机相连，该交换机通过一个或者两个 gigabit 的上行链路与一个核心的 gigabit 交换机相连，该核心交换机连接所有的机柜。

我们的根本选择标准是单次查询花费，可以表示为性能/资金花费总和(包括折旧)+管理花费(hosting，系统管理，维修)。实际来看，一个服务器的寿命通常不会超过 2，3 年，因为与新机器相比，无法在性能上保持一致。三年前的机子性能上要远远落后于当前的机子，对于包含这两类机器的集群，很难达到合适的负载分布和配置。有了这个相对的短期分摊周期，可以看到设备的花费在总的开销中占到相当的一部分。

google 服务器是定制的，我们可以使用一个具有可比性的基于 pc 的服务器机柜价格做一个说明。比如 ，在 2002 年末，一个 由 88 个具有双核 cpu 2G intle xeon，2G 内存，80G 硬盘的机器组成的机柜在 RackSaver.com 上的价格大概是 27800$,转换成三年周期每月的花费将是 7700$。剩下的主要花费是人力和 hosting。

设备开销的相对重要性使得采用传统的服务器解决方案并不适合解决我们的问题，因为虽然它们可以提高性能，但是降低了性价比。比如 4 核处理器的主板是很昂贵的，由于我们的应用已经进行了很好的并行化，这样的主板虽然性能好但是对我们来说就是得不偿失了。类似的，尽管 SCSI 硬盘更快也更可靠，但是它们通常比同样容量的 IDE 硬盘贵 2-3 倍。

使用基于廉价 PC 的集群比使用高端多处理器服务器在成本上的优势是非常明显的，至少对于像我们这样的高并行化应用来说。上面例子中的 27800$的机柜包含 176 个 2G Xeon CPU，176G 内存，7T 硬盘空间。与此相比，一个典型的 x86 服务器具有 8 个 2GCPU，64G 内存，8T 硬盘空间，花费大概 758000$。换句话说，这个服务器贵了 3 倍，但是 cpu 只是原来的 1/22，内存是 1/3，硬盘空间稍微多了点。价格的差距，主要是源于更高的互联网络带宽和可靠性。但是 google 的

高度冗余架构并不依赖于这两个中的任何一个。

管理数千台中型 PC 机和一些高端多处理器服务器将会带来完全不同的系统管理和维修费用。然而，对于一个相对同构的的应用来说，大部分的服务器只用来运行有限的应用中的某一个，这些开销是可管理的。假设安装和更新工具都是可用的，维护 1000 台和 100 台服务器所需的时间和成本相差并不大，因为所有的机器都具有相同的配置。类似的，通过使用可扩展的应用监控系统，监控的花费伴随着集群大小的增长也不会有太大的增长。另外，我们可以通过批处理修复，将修复的开销保持在一个较低的水平，同时保证我们可以很容易的替换掉那些具有高损坏率的组件，比如硬盘和电源。

电力问题

如果没有特殊的高密度包装，电力消耗和制冷设备将会成为一个挑战。一个中型的 1.4G 奔三服务器在有负载情况下，通常要耗费 90 瓦直流电：55w 给 cpu，10w 给硬盘，25w 给 DRAM 加电和主板，对于一个 ATX 电源，通常具有 75%的效率，这样转化成交流电就是 120 瓦，每个机柜就需要 10 千瓦。一个机柜需要 25 平方英尺 的空间，这样算下来，用电的密度就是 400 瓦/平方英尺。如果采用高端处理器，一个机柜的用电密度会超过 700 瓦/平方英尺。

不幸的是，通常的商业数据中心电力密度在 70-150 瓦/平方英尺之间，远远低于 pc 集群的需求。这样，如果低端 pc 集群使用相对直接的包装方式，就需要特殊的制冷和额外的空间来降低用电密度使得与标准数据中心兼容。因此只要机柜还存放在标准数据中心中，如果想要往机柜里增加服务器就会在实际部署中受到限制。这种情况就使得我们考虑降低单服务器的电力使用是否是可能的。

对于大规模集群来说，低功耗服务器是非常具有吸引力的。但是我们必须要牢记以下几点：降低电力是迫切的，但是对于我们的应用来说，不能带来性能上的惩罚，我们关心的是瓦特性能比，不是单纯的瓦特，第二，低功耗的服务器必须不能太过昂贵，因为服务器的折旧花费通常要超过电力的花费。前面提到的 10kw 机柜，每月大概消耗 10mwh 的电力(包括制冷的开销)，假设每 kwh 电 15 美分，这样每月需要花费 1500$，与折旧的 7700$相比并不算大。因此为了保证总体的成本优势，低功耗服务器不能够比采用常规 pc 更昂贵{!准确的说应该是要保证(比较低功耗服务器本身的价格-节省的电力开销成本<=常规 PC 价格)}。

硬件级别的应用特点

检查我们应用的各种架构上特点可以帮助我们搞清楚哪种硬件平台可以为我们的索引查询系统提供最高的性价比。我们着重于索引服务器的特点，该模块的性

价比对于整体的性价比起着主导性的作用。索引服务器的任务主要包括：对倒排索引中的压缩数据进行解压，找到与一个查询相匹配的文档集合。表 1 展示了一些索引服务器程序的基本的指令级别的度量，程序运行在一个 1G 双核奔三系统上。

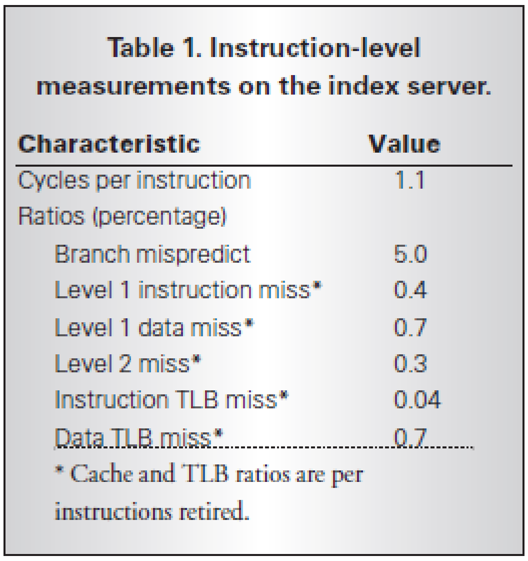


Table 1. Instruction-level measurements on the index server.

Characteristic	Value
Cycles per instruction	1.1
Ratios (percentage)	
Branch mispredict	5.0
Level 1 instruction miss*	0.4
Level 1 data miss*	0.7
Level 2 miss*	0.3
Instruction TLB miss*	0.04
Data TLB miss*	0.7

* Cache and TLB ratios are per instructions retired.

考虑到奔三每个指令周期基本上可以处理三条指令，应用程序具有一个适度的高 CPI(cycles per instruction)稍微偏高。同时考虑到我们的应用程序使用了很多动态数据结构而控制流是依赖于数据的，这样就会产生大量的很难预测的分支。事实上，如果相同的工作负载在奔四处理器上运行时，CPI 几乎增长了 2 倍，但分支预测几乎相同，尽管奔四具有更强大的指令并行和分支预测功能。可见，在这种工作负载类型下，并没有太多的指令级并行性可供挖掘。测量结果显示，对于我们的应用来说，出现在处理器中的大量的乱序不确定性执行成为降低程序性能的关键点。

(A more profitable way to exploit parallelism for applications such as the index server is to leverage the trivially parallelizable computation.)对于像索引服务器这样的应用来说，更合适的挖掘并行性的方式应该是提高平凡的计算并行性。(Processing each query shares mostly readonly data with the rest of the system and constitutes a work unit that requires little communication.)系统在处理每个查询时共享只读数据，建立只需要很少通信的工作单元。我们已通过在集群级别上部署大量的廉价节点取代少量的昂贵节点来发挥这个优势。挖掘在微架构级别上的线程级并行性看起来也是可行的。并行多线程(SMT)和多处理器架构(CMP)都是面向线程级的并行性，都可以大大提高我们的服务器的性能。一些针对Intel Xeon处理器的早期实验表明通过使用两个上下文的SMT比单个上下文具有30%的性能提升。

我们相信对于 CMP 系统提升的潜力应该是更大的。在 CMP 的设计中，采用多个简单的，按序执行的，短流水线核取代复杂的高性能核。如果我们的应用具有很少的指令级并行性(ILP)，那么由于按序执行所带来的性能惩罚也是很小的。同时短流水线将可以减少甚至排除分支预测失败所造成的影响。线程级的并行性伴随着核的增长可以呈现出接近线性的加速比，同时一个合理大小的共享 L2 cache

将会加速处理器间的通信。

内存系统

表 1 也描述了主存系统的性能参数，我们可以观察到对于指令 cache 和指令 tlb 具有良好的性能，由于使用了较小的内层循环代码。索引数据块不具有时间局部性，因为索引数据大小变化剧烈同时对于索引的数据块访问模式是不可预测的。然而对于一个索引数据块的访问可以从空间局部性上获益，这种局部性能够通过硬件预取或者大的缓存 line 开拓出来。这样如果使用相对合适的 cache 大小就可以得到好的全局 cache 命中率。

内存带宽看起来并不会成为一个瓶颈。我们估计奔腾系列处理器系统的内存带宽使用率可以很好的控制在 20%以下。主要是由于对于索引数据的每个缓存行，需要放到处理器 cache 里，这需要大量的计算资源，此外在数据获取中还存在天然的依赖关系。在很多情况下，索引服务器的内存系统行为正如 TPC-D(Transaction Processing Performance Counicil’s benchmark D)所报告的那样。对于这种工作负载类型，采用一个相对合适的 L2cache 大小，短的 L2 cache 和内存延时，长的(比如 128 字节)cache line 可能是最有效的。

大规模多处理(Large-scale Multiprocessing)

正如前面提到的，我们的设备是一个由大量廉价 pc 组成的庞大集群，而不是少数大规模的共享内存机组成的。大规模共享内存机主要用于在计算通信比很低的时候，通信模式或者数据划分是动态或者难预测的，或者总的花费使得硬件花费显得很少的时候(由于管理日常费用和软件许可证价格)。在这些情况下，使得它们的高价格变得合理。

在 google，并不存在这样的需求，因为我们通过划分索引数据和计算来最小化通信和服务器间的负载平衡。我们自己开发需要的软件，通过可扩展的自动化和监控来降低系统管理的日常费用，这些使得硬件花费成为整个系统开销中显著的一块。另外，大规模的共享内存机器不能很好的处理硬件或者软件的失败。这样大部分的错误可能导致整个系统的 crash。通过部署大量的多处理器机，我们可以将错误的影响控制在一个小的范围内。总的来看，这样的一个具有显著的低成本的集群解决了我们的服务对于性能和可用性的需求。

初看起来，好像很少有应用具有像 google 这样的特点，因为很少有服务需要数千台的服务器和数 pb 的存储。然而可能有很多的应用需要使用基于 pc 的集群架构。一个应用如果关注性价比，能够运行在不具有私有状态的多个服务器上(这样服务器就可以被复制)，它都有可能从类似的架构中获益。比如一个高容量的

web 服务器或者一个计算密集型的应用服务器(必须是无状态的)。这些应用具有大量的请求级并行性(请求可以划分在在独立的服务器上处理)。事实上，大的 web 站点已经采用这样的架构。

在 google 的应用规模上，大规模服务器的并行化的一些限制确实变得明显起来，比如商业数据中心在制冷容量上的限制，当前的 cpu 对于面向吞吐率的应用所做的优化还远远不够。虽然如此，通过使用廉价 pc，明显地提高了我们可以为单个查询所能支付的计算量。因此有助于帮助我们提高成千上万的用户的网络搜索体验。

致谢

在这些年里，很多人为 google 的硬件架构做出了重要的贡献。在这里，特别感谢 Gerald Aigner ,Ross Biro,Bogdan Cocosel 和 Larry Page 所做的工作。

Google 文件系统

作者：Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung Google Inc 2003
译者：phylips@bmy 2010-10-1
出处：http://duanple.blog.163.com/blog/static/7097176720109145829346/

摘要

我们设计实现了 google 文件系统，一个面向大规模分布式数据密集性应用的可扩展分布式文件系统。它运行在廉价的商品化硬件上提供容错功能，为大量的客户端提供高的整体性能。

尽管与现有的分布式文件系统具有很多相同的目标，我们的设计更多的来源于对于我们的具体应用的负载类型以及当前甚至未来技术环境的观察，这就使得它与早期的文件系统表现出明显的不同。这也使得我们重新审视传统的设计选择，探索出一些与之根本不同的设计观点。

这个文件系统成功的满足了我们的存储需求。随着研究和开发人员的努力，在 google 内部，它已经作为基础存储平台而广泛部署，服务于那些具有海量数据生成与处理需求的服务。迄今为止，最大的集群已可以通过一千多台机器上的数千块硬盘提供数百 T 的存储，这些存储空间可以由数百个客户端并发访问。

在本论文中，我们将描述用于支持分布式应用的文件系统接口扩展，讨论很多我们的设计观点，展示来自于 beachmark 和现实世界的一些测量数据。

分类和主题描述
分布式文件系统

常用词
设计，可靠性，性能，测量

关键词
容错，可扩展，数据存储，集群存储

导引

为了满足 google 快速增长的数据处理需求，我们设计实现了 google 文件系统(GFS)。GFS 与传统的分布式文件系统具有很多相同的目标比如性能、可扩展性、可靠性、可用性。然而，它的设计是由我们的具体应用的负载类型以及当前甚至未来技术环境的观察驱动的，所以与早期文件系统的设计假设具有明显的区别。我们重新

审视传统上的设计选择，探索出一些在根本上不同的设计观点。

一、组件失败成为一种常态而不是异常。文件系统是由成百上千台通过廉价的商品化部件组装起来的存储机器构成，可以被大量的客户端访问。组件的数量和质量在本质上决定了在某一时间有一些是不可用的，有一些无法从当前的失败中恢复过来。我们观察到，应用程序的 bug、操作系统 bug、人为的错误、硬盘的失败、内存、连接器(connectors)、网络、电力供应都可以引起这样的问题。因此常规性的监控、错误检测、容错和自动恢复必须集成到系统中。

二、与传统标准相比，文件是巨大的。在这里，好几个 G 的文件是很普通的。每个文件通常包含应用程序处理的多个对象比如网页文档。当我们日常处理的快速增长的数据集合总是达到好几个 TB 的大小，包含数十亿的对象时，去处理数十亿个 KB 级别的文件即使文件系统支持也会显得很笨重。这样设计中的一些假设和参数，比如 IO 操作和块大小就必须重新定义。

三、大部分的文件更新模式是通过在尾部追加数据而不是覆盖现有数据。文件内部的随机写操作几乎是不存在的。一旦写完，文件就是只读的，而且通常是顺序读。大量的数据都具有这样的特点：它们有些可能是会被数据分析程序扫描的主体，有些可能是由运行中的应用程序持续生成的数据流，有些可能是档案数据，有些可能是由某台机器处理产生的中间结果，然后 会由另一台机器并行或者稍后再进行处理。在这种大文件的数据访问模式下，append 操作就成为性能优化和原子性保证的关键，而在客户端缓存数据块就显得不那么重要了。

四、应用程序和文件系统 API 的协同设计，增加了整个系统的灵活性。比如我们通过放松了 GFS 的一致性模型大大简化了文件系统，同时也没有给应用程序带来繁重的负担。我们也提供了一个原子性的 append 操作，这样多个客户端就可以对同一个文件并行的进行 append 操作而不需要彼此间进行额外的同步操作。这些都会在后面进行详细的讨论。

目前已经有多个 GFS 集群为了不同的目的而部署起来。最大的具有 1000 个存储节点，超过 300T 的磁盘空间，持续地为来自不同机器的数百个客户端提供服务。

2 设计概览

2.1 假设

在为我们的需求设计一个文件系统过程中，一些充满了挑战和机遇的假设指导着我们最终的设计，之前我们已提到过一些关键点，现在我们把这些假设再详细的列出来。

- 系统是由廉价的经常失败的商品化组件构建而来。必须进行常规性的监控和检测，容错，并且能够从组件失败中迅速的恢复，这些都应该像是例行公事。
- 系统存储了适度个数的大文件。我们期望有数百万个文件，每个 100MB 或

者更大。上 GB 的文件大小应该是很普通的情况而且能被有效的管理。小文件也应该被支持，但我们不需要为它们进行优化。

- 工作负载主要由两种类型的读组成：大的顺序流式读取和小的随机读取。在大的流式读取中，单个操作通常要读取数百 k，甚至 1m 或者更大的数据。来自于同一个客户端的连续读取，通常读完文件的一个连续区域。小的随机读取通常在某个任意的偏移位置读取几 kb 的数据。应用程序通常都很注重性能，会把小的随机读取批量处理，并进行排序使得读取可以稳步的穿越整个文件而不是来回读取。
- 工作负载有很多大的对文件数据的 append 操作，通常操作的数据大小类似于读操作。一旦写完，就很少改变，可以支持在文件内部的随机写操作，但是性能不必很高。
- 系统必须为多个客户端对相同文件并行 append 操作的定义良好(well-defined)的语义提供有效的实现。文件通常是用来作为生产者消费者队列或者进行多路归并。数百个生产者(每个机器上运行一个)将会对同一个文件进行 append。因此具有最小化同步开销的原子性是必要的。文件可能之后才会被读取，或者某个消费者可能正在并行读取这个文件。

- 持续的高带宽比低延时更重要。我们大部分的目标应用都希望得到高速的批量数据处理速度，很少有对于单个的读写有严格的响应时间需求。

2.2 接口

GFS 提供一个熟悉的文件系统接口，尽管它并没有实现一个诸如 POSIX 那样的标准 API。文件通过目录进行层次化组织，并通过路径名来标识文件。支持常见的那些文件操作：create,delete,open,close,read,write。

另外，GFS 还具有快照和 append 操作。快照以很低的开销创建一个文件或者目录树的拷贝。append 操作允许多个客户端向同一个文件并发的操作，同时保证每个独立客户端的 append 操作的原子性。这对于实现结果的多路归并以及生产者消费者队列很有帮助，这样客户端不需要额外的锁操作就可以并行的 append。我们发现这种文件类型，对于构建大型的分布式应用简直就是无价之宝。快照和 append 操作将会在 3.4 和 3.3 分别讨论。

2.3 架构

一个 GFS 集群由一个 master 和多个 chunkserver 组成，可以被多个 client 访问，如图 1 所示。

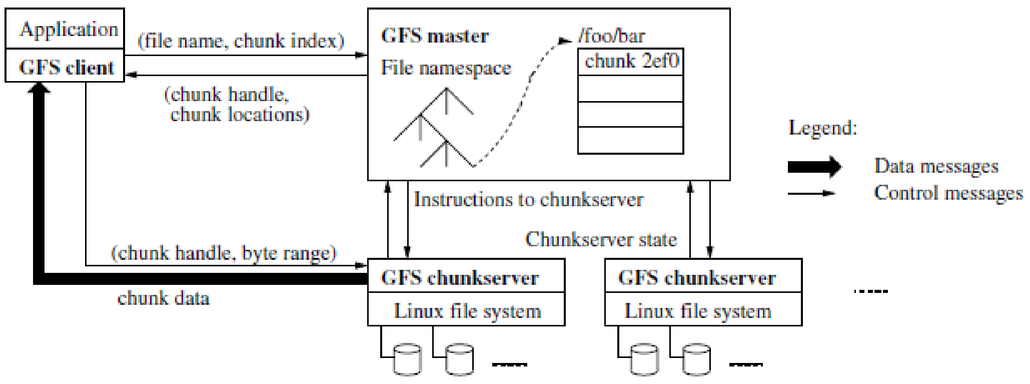


Figure 1: GFS Architecture

它们都是普通的 linux 机器，上面运行着一些用户级服务进程。可以很容易的在同一台机器上运行一个 chunkserver 和 client，只要机器资源允许以及由运行各种古怪的应用程序代码带来的低可靠性是可以接受的。

文件被划分成固定大小的 chunk。每个 chunk 是由 chunk 创建时由 master 分配的一个不可变的全局唯一的 64bit 句柄来标识。Chunkserver 将 chunk 作为 linux 文件存储在本地，对于 chunk 数据的读写通过 chunk 的 handle 和字节边界来表示。为了可靠性，每个 chunk 存储在多个 chunkserver 上。尽管用户可以为不同文件名字空间区域指定不同的备份级别，默认地我们存储三个备份。

Master 维护所有的文件系统元数据。包括名字空间，访问控制信息，文件与 chunk 的映射信息，chunk 的当前位置。它也控制系统范围内的一些活动，比如 chunk 租约管理，无效 chunk 的垃圾回收，chunkserver 间的 chunk 迁移。Master 与 chunkserver 通过心跳信息进行周期性的通信，以发送指令和收集 chunkserver 的状态。

应用程序链接的 GFS 客户端代码实现了文件系统 API，应用程序通过它与 master 和 chunkserver 进行通信以读写数据。客户端如果需要操作元数据则需要与 master 通信，但是所有的纯数据通信直接与 chunksever 通信。我们没有提供 POSIX API，因此也就没有必要将它与 linux vnode 层关联。

客户端或者 chunkserver 都不会进行文件数据缓存。客户端缓存只能得到很少的好处，因为大部分的应用需要直接读取整个大文件或者工作集合太大根本无法缓存。没有 cache 简化了客户端和整个系统，因为不需要考虑缓存一致性问题(实际上客户端会缓存元数据)。Chunkserver 不需要进行文件数据缓存，是因为 chunk 是作为本地文件存储，这样 linux 自身会将那些经常访问的数据进行缓存。

2.4 单 Master

只有一个 master 大大简化了我们的设计，而且使得 master 可以利用全局信息对 chunk 的放置和备份进行更好的判断。然而，我们必须最小化它在读写中的参与度，使得它不会成为一个瓶颈。Client 永远不会通过 master 读取文件数据，它只

是问 master 它应该同哪个 chunkserver 联系。并且 client 会在一定的时间段内缓存这些信息，直接与 chunksever 交互进行很多后续的操作。

根据图 1，我们解释一下一个简单的读操作的交互过程：首先，由于 chunk 的大小固定，客户端就可以将应用程序中标识的文件名和 offset 转换为 chunk 的 index。然后给 master 发送一个包含文件名和 chunk index 的请求，master 返回相应的 chunk 的 handle 和所有备份的位置。客户端以文件名和 chunk index 为 key 将这条信息进行缓存。

然后客户端给其中一个备份发送一个请求，通常是最近的那个。请求标识了 chunk 的 handle 以及在那个 chunk 内的字节边界。直到缓存信息过期或者重新打开文件之前，对于相同 chunk 的后续读操作就不需要 client-master 的通信了。事实上，客户端通常在一个请求中查询多个 chunk 的信息，master 也可以将这些被请求的多个 chunk 的信息包裹在一块进行返回。通过使用这种特别的信息，没有增加额外的花费就避免了未来 client-master 间的多次通信。

2.5 chunk 大小

chunk 大小是一个关键的设计参数。我们选择了 64MB，远远大于现有的文件系统块。每个 chunk 的副本作为普通的 linux 文件存储在 chunkserver 上，只在需要时才会进行扩展。惰性空间分配策略避免了因内部碎片造成的空间浪费，很可能最大的碎片有像一个 chunk 那么大。

大的 chunk size 提供了几个重要的优势。首先，降低了 client 与 master 的交互需求，因为在相同 chunk 上的读写只需要一个初始化请求就可以从 master 得到 chunk 的位置信息。这对于减少应用产生的负载是非常明显的，因为大部分应用需要顺序的读写整个大文件。即使对于小的随机读取，客户端也可以很容易的缓存一个几 TB 工作集的所有 chunk 的位置信息。其次，由于 chunk 很大，那么客户端就很有可能在一个给定的 chunk 上执行更多的操作，这样可以将一个与 chunkserver 的 TCP 连接保持更长的时间，这就减少了网络开销。再者，降低了存储在 master 上的元数据大小。这样就允许我们将元数据存放在内存中，反过来就带来了我们将在 2.6.1 中讨论的其他优势。

另一方面，大的 chunk size，即使采用了 lazy 空间分配，也有它的缺点。小的文件可能只有少数几个 chunk，或许只有一个。如果很多的 client 都需要访问这个文件，这样那些存储了这些 chunk 的 chunkserver 就会变成热点。实际中，热点还没有成为一个主要的考虑点因为我们的应用绝大部分都是在顺序读很大的多 chunk 文件。

然而，当 GFS 第一次使用在一个批处理队列系统时，热点确实出现了：一个可执行文件作为只有一个 chunk 的文件写到 GFS，然后同时在数百台机器上开始执行。存储了该可执行文件的那些 chunkserver 被数百个并发请求瞬间变成超载。我们通过更高的备份级别存储这样的可执行文件以及减慢队列系统的应用程序启动时间解决了这个问题。一个潜在的长远解决方案是在这种情况下，允许客户端从

其他客户端读取数据。

2.6 元数据

Master 存储了三个主要类型的元数据：文件和 chunk 名字空间，文件到 chunk 的映射信息，每个 chunk 的备份的位置。所有的元数据都保存在 master 的内存中。前两种类型还通过将更新操作的日志保存在本地硬盘和备份在远程机器来保持持久化。使用 log 允许我们简单可靠地更新 master 的状态，不用担心当 master crash 时的不一致性。Master 并没有永久保存 chunk 的位置信息，而是在 master 启动或者某个 chunkserver 加入集群时，它会向每个 chunkserver 询问它的 chunks 信息。

2.6.1 内存数据结构

由于元数据存储在内存里，master 的操作是很快的。因此对于 master 来说，可以简单有效地在后台对整个状态进行周期性扫描。这个周期性的扫描是用来实现 chunk 垃圾回收，chunkserver 出现失败时进行的重复制，以及为了平衡负载和磁盘空间在 chunkserver 间的 chunk 迁移。4.3，4.4 将进一步讨论这些活动。

全内存策略存在的一个潜在限制就是 chunk 的数目，因此整个系统的容量取决于 master 有多少可用内存。实际中这不是一个很严重的限制。Master 为每个 64MB 的 chunk 维护少于 64byte 的数据。大部分的 chunk 是满的，因为大部分的文件包含多个 chunk，只有最后一个 chunk 可能是未满的。类似的，每个文件名字空间数据通常需要少于 64byte 因为文件名称存储时会使用前缀压缩算法进行压缩。

如果需要支持更大的文件系统，只需要往 master 里添加内存。这点开销与通过将元数据存储到内存所得到简单性、可靠性、性能和灵活性相比，将是很小的一笔花费。

2.6.2 chunk location

Master 并没有提供一个永久性的存储保存对于一个给定的 chunk 都是那些 chunkserver 保存了它的副本。它只是在启动时，简单地从 chunkserver 那里把这些信息拉过来。Master 能够保证它自己是更新过的，因为是由它来控制 chunk 的放置，以及通过周期性的心跳信息来监控 chunkserver。

起初，我们尝试将 chunk 位置信息永久保存在 master，但是我们发现在启动时去 chunkserver 请求这些数据更简单。这样避免了当 chunkserver 在加入或者离开集群，改名，失败，重启等待时需要的 master 与 chunkserver 间的同步。在一个数百台机器的集群中，这样的事件太经常了。

理解这个设计决定的另一个方式是 chunkserver 对于自己有还是没有某个 chunk 具有最终的发言权。在 master 上维护一个这些信息一致性视图是没有意义的，

因为发生在 chunkserver 上的错误可能使得一些 chunk 突然间不见了(比如硬盘可能会坏掉或者不可用)，一个操作可能将 chunkserver 重命名。

2.6.3 操作日志

操作日志包含了关键元数据改变的历史记录。它是 GFS 的核心。它不仅是元数据的唯一一致性记录，而且它也定义了那些并发操作的逻辑上的时间表。文件和 chunk 的版本都是唯一和永恒地由它们创建时的逻辑时间来标识的。

因此操作日志是很关键的，我们必须可靠地保存它，在任何元数据变更被持久化之前不应当被客户端看到。否则，我们将丢失整个文件系统或者最近的客户端操作，即使 chunckserver 自己保存了它们。因此我们将它备份在多个远程机器上，对于一个客户端操作只有当该操作对应的日志记录被刷新到本地和远程的磁盘上时才会发出响应。Master 将几个操作日志捆在一块刷新，从而降低刷新和复制对于整个系统吞吐率的影响。

Master 通过重新执行操作日志来恢复它的文件系统。为了最小化启动时间，我们必须将日志保持在很小的规模。当日志增长超过一定的大小后，Master 给它的状态设置检查点，它可以通过从本地磁盘加载最新的检查点进行恢复，然后重新执行那些在该检查点之后的日志记录。检查点保存了一个压缩的类 B 树的结构，不需要额外的解析就可以直接映射到内存用于名字空间查找。这大大提高了恢复的速度和可用性。

因为建立一个检查点会花费一些时间，master 内部状态结构的设计使得一个新的检查点可以不需要延时那些接受到的变化就可以被创建。Master 会启动一个新的线程切换到一个新的日志文件然后创建新的检查点。这个新的检查点包含在切换之前的所有变更。对于一个包含几百万文件的集群大概需要几分钟就可以完成。结束后，它将会被写回本地和远程的磁盘。

恢复只需要最新的检查点和后来的日志文件。更老的检查点和日志文件可以自由的删除，当然我们会保存一些来应对某些突发情况。在创建检查点的时候发生的失败不会影响系统的正确性，因为恢复代码会检测和跳过不完全的检查点。

2.7 一致性模型

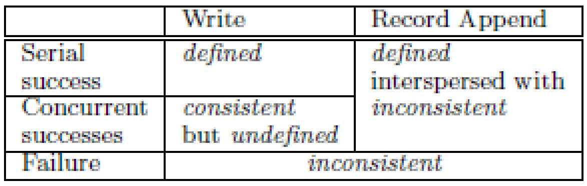
GFS 使用了一个放松的一致性模型不但很好的支持了我们的高度分布式的应用，而且实现起来也相对简单高效。我们现在讨论 GFS 所提供的保证以及它们对应用程序意味着什么。我们也会讲述 GFS 如何维护这些保证，但是会将具体的细节留到其他论文里讲述。

2.7.1 GFS 提供的保证

文件名字空间的改变(比如文件创建)是原子性的。它们只由 master 进行处理：名

字空间锁用来保证原子性和正确性(4.1 节)。Master 的操作日志定义了这些操作的全局性的顺序。

当数据变更后，文件区域的状态取决于变更的类型，变更是否成功以及是否是并发进行的。表 1 是对变更结果的一个概述。



	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

Table 1: File Region State After Mutation

如果所有的客户端无论从哪个副本读取数据总是看到相同的数据，那么我们就说文件区域是一致的(consistent)。如果文件数据变更后是一致的，同时客户端可以看到它所有的变更，那么我们就说文件区是已定义的(defined)。当一个变更成功后，且没有受到其他并发写者的影响，那么被影响的区域就是已定义的(defined)(肯定是一致性的)：所有的客户端总是能看到该变更所写入的数据。并发的成功的变更，会使区域进入未定义的状态但是还是一致的：所有的客户端可以看到一致的数据，但是它可能无法看到所有的变更{！如果变更是针对相同的数据写这样有的变更就会被新的变更所覆盖，这样用户就无法看到最先的变更了，同时发生在跨 chunk 的操作会被拆分成两个操作，这样这个操作的一部分可能会被其他操作覆盖，而另一部分则保留下来，如 3.1 节末尾所述}。通常它看到的是多个变更组合后的结果。一个失败的变更会使区域进入非一致的状态(因此也是未定义的状态)：不同的客户端在不同的访问中可能看到不同的数据。我们后面会描述我们的应用程序如何区分已定义的(defined)区域和未定义的(undefined)区域。应用程序不需要进一步区分未定义区域的各种不同类型。

数据变更可能是写或者记录 append。写操作会使数据在应用程序指定的偏移位置写入。记录 append 操作会使数据原子性的 append，如果是并发性的话则至少会被 append 一次{！应该是指每个 append 操作会至少执行一次，而不是说至少有一个 append 操作会执行，比如多个 client 同时发起 append 操作，它们每个都应该会至少 append 一次，而非至少有一个 client 成功 append}，但是 偏移位置是由 GFS 决定的(然而，通常的理解可能是在客户端想写入的那个文件的尾部)。偏移位置会被返回给客户端，同时标记包含这条记录的那个已定义的(defined)文件区域的起始位置。另外 GFS 可能会在它们之间插入一些 padding 或者记录的副本。它们会占据那些被认为是不一致的区域，通常它们比用户数据小的多。

在一系列成功的变更之后，变更的文件区域被保证是已定义的(defined)，同时包含了最后一次变更的数据写入。GFS 通过两种方式来实现这种结果 a.将这些变更以相同的操作顺序应用在该 chunk 的所有的副本上，b.使用 chunk 的版本号来检测那些可能由于它的 chunkserver 挂掉了而丢失了一些变更的陈旧副本。陈旧的副本永远都不会参与变更或者返回给那些向 master 询问 chunk 位置的 client。它们会优先参与垃圾回收。

因为客户端会缓存 chunk 的位置，在信息更新之前它们可能会读到陈旧的副本。该时间窗口由缓存值的超时时间以及文件的下一次打开决定，它们会清除缓存中该文件相关的所有 chunk 信息。此外，由于我们的大部分操作都是记录的 append，因此一个陈旧副本通常会返回一个过早结束的 chunk 而不是过时的数据。当读取者重试并与 master 联系时，它会立即得到当前的 chunk 位置。

在一个成功的变更发生很久之后，组件失败仍有可能破坏或者污染数据。GFS 通过周期性的 master 和所有 chunkserver 间的握手找到那些失败的 chunkserver，同时通过校验和(5.2 节)来检测数据的污染。一旦发现问题，会尽快地利用正确的副本进行恢复(4.3 节)。只有一个块的所有副本在 GFS 做出反应之前全部丢失，这个块丢失才是不可逆转的，而通常 GFS 的反应是在几分钟内的。即使在这种情况下，块不可用，而不是被污染：应用程序会收到清晰的错误信息而不是被污染的数据。

2.7.2 对于应用程序的影响

GFS 应用程序可以通过使用简单的技术来适应这种放松的一致性模型，这些技术已经为其他目的所需要：依赖于 append 操作而不是覆盖、检查点、写时自我验证、自标识-记录。

实际中，我们所有的应用程序都是通过 append 而不是覆盖来改变文件。在一个典型应用中，一个写操作者会从头至尾生成一个文件。当写完所有数据后它自动的将文件重命名为一个永久性的名称，或者通过周期性的检查点检查已经有多少数据被成功写入了。检查点可能会设置应用级的校验和。读取者仅验证和处理最后一个检查点之前的文件区域，这些区域处于已定义的状态。无论什么样的并发和一致性要求，这个方法都工作的很好。append 操作比随机写对于应用程序的失败处理起来总是要更加有效和富有弹性。检查点允许写操作者增量性的重启(不需要重新从头写)，允许读取者可以处理那些已经成功写入的数据，虽然应用程序看到的数据仍然是不完全的。

另一种典型的应用中，很多写者为了归并文件或者是作为一个生产者消费者队列同时向一个文件 append。记录的 append 的 append-at-least-once 语义保证了每个写者的输出。只是读取者需要采用如下方法处理偶然的 padding 和重复数据。写者为每条记录准备一些额外信息比如校验和，这样它的合法性就可以验证，通过校验和，读取者就可以识别并且忽略掉这些冗余的 padding 和记录片段。如果不能容忍重复的数据(比如它们可能触发非幂等操作)，可以通过在记录中使用唯一标识符来过滤它们，很多时候都需要这些标识符命名相应的应用程序实体，比如网页文档。这些用于 record 输入输出的功能函数是以库的形式被我们的应用程序共享的，同时应用于 gongle 其他的文件接口实现。所以，总是可以传送给记录读取者相同系列的记录，加上一些很少的重复数据。

在以上的描述中，存在一个基本的假定：数据是以 record 形式存储的，而且通常这些 record 都是可以重复的，比如一个网页文档我们可以重复存，这对于数

百亿的网页文档来说，存储少数多余的很正常，也就是说这些数据通常是文本，而不是二进制，所以我们才可以在 append 或者写时用记录的副本来覆盖非一致的区域，所以提供了 append 的 append-at-least-once 语义，因为 append 两次也是可以的。如果我们要保证唯一性，可以在应用层增加逻辑。

3.系统交互

我们是以尽量最小化 master 在所有操作中的参与度来设计系统的。在这个背景下，我们现在描述下 client，master 以及 chunkserver 之间是如何交互来实现数据变更、记录 append 以及快照的。

3.1 租约和变更顺序

一个变更是指一个改变 chunk 的内容或者元信息的操作，比如写操作或者 append 操作。每个变更都需要在所有的副本上执行。我们使用租约机制来保持多个副本间变更顺序的一致性。Master 授权给其中的一个副本一个该 chunk 的租约，我们把它叫做主副本(primary)。这个主副本为针对该 chunk 的所有的变更选择一个执行顺序，然后所有的副本根据这个顺序执行变更。因此，全局的变更顺序首先是由 master 选择的租约授权顺序来确定的(可能有多个 chunk 需要进行修改)，而同一个租约内的变更顺序则是由那个主副本来定义的。

租约机制是为了最小化 master 的管理开销而设计的。一个租约有一个初始化为 60s 的超时时间设置。然而只要这个 chunk 正在变更，那个主副本就可以向 master 请求延长租约。这些请求和授权通常是与 master 和 chunkserver 间的心跳信息一起发送的。有时候 master 可能想在租约过期前撤销它(比如，master 可能想使对一个正在重命名的文件上的变更无效)。即使 master 无法与主副本进行通信，它也可以在旧的租约过期后安全的将租约授权给另一个新的副本。

如图 2，我们将用如下的数字标识的步骤来表示一个写操作的控制流程。
1.client 向 master 询问哪个 chunkserver 获取了当前 chunk 的租约以及其他副本所在的位置。如果没有人得到租约，master 将租约授权给它选择的一个副本。
2.master 返回该主副本的标识符以及其他副本的位置。Client 为未来的变更缓存这个数据。只有当主副本没有响应或者租约到期时它才需要与 master 联系。
3.client 将数据推送给所有的副本，client 可以以任意的顺序进行推送。每个 chunkserver 会将数据存放在内部的 LRU buffer 里，直到数据被使用或者过期。通过将控制流与数据流分离
，我们可以通过将昂贵的数据流基于网络拓扑进行调度来提高性能，而不用考虑哪个 chunkserver 是主副本。3.2 节更深入地讨论了这点。
4.一旦所有的副本接受到了数据，client 发送一个写请求给主副本，这个请求标识了先前推送给所有副本的数据。主副本会给它收到的所有变更(可能来自多个 client)安排一个连续的序列号来进行必需的串行化。它将这些变更根据序列号应用在本地副本上。

5.主副本将写请求发送给所有的次副本，每个次副本以与主副本相同的串行化顺序应用这些变更。
6.所有的次副本完成操作后向主副本返回应答
7.主副本向 client 返回应答。任何副本碰到的错误都会返回给 client。出现错误时，该写操作可能已经在主副本以及一部分次副本上执行成功。(如果主副本失败，那么它不会安排一个序列号并且发送给其他人)。客户端请求将会被认为是失败的，被修改的区域将会处在非一致状态下。我们的客户端代码会通过重试变更来处理这样的错误。它会首先在 3-7 步骤间进行一些尝试后在重新从头重试这个写操作。

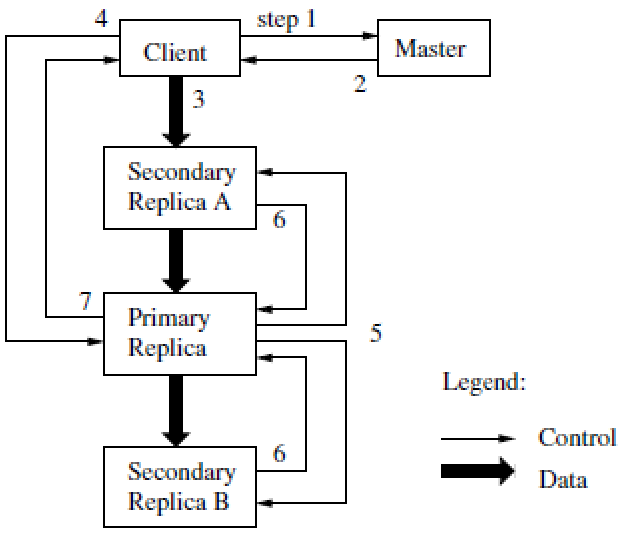


Figure 2: Write Control and Data Flow

如果应用程序的一个写操作很大或者跨越了 chunk 的边界，GFS client 代码会将它转化为多个写操作。它们都会遵循上面的控制流程，但是可能会被来自其他 client 的操作插入或者覆盖。因此共享的文件区域可能会包含来自不同 client 的片段，虽然这些副本是一致的，因为所有的操作都按照相同的顺序在所有副本上执行成功了。但是文件区域会处在一种一致但是未定义的状态，正如 2.7 节描述的那样。

3.2 数据流

为了更有效的使用网络，我们将数据流和控制流分离。控制流从 client 到达主副本，然后到达其他的所有次副本，而数据则是线性地通过一个仔细选择的 chunkserver 链像流水线那样推送过去的。我们的目标是充分利用每个机器的网络带宽、避免网络瓶颈和高延时链路，最小化数据推送的延时。

为了充分利用每个机器的网络带宽，数据通过 chunkserver 链线性的推送过去而不是以其他的拓扑进行分布比如树型。因此每个机器的全部输出带宽可以用来尽快地发送数据而不是划分给多个接受者。

为了尽可能的避免网络瓶颈和高延时链路，每个机器向网络中还没有收到该数据的最近的那个机器推送数据。假设 client 将数据推送给 S1- S4，它会首先将数据推送给最近的 chunkserver 假设是 S1，S1 推送给最近的，假设是 S2，S2 推送给 S3、S4 中离他最近的那个。我们网络拓扑足够简单，以至于距离可以通过 IP 地址估计出来。

最后为了最小化延时，我们将 TCP 数据传输进行流水化。一旦一个 chunkserver 收到数据，它就开始立即往下发送数据。流水线对我们来说尤其有用，因为我们使用了一个全双工链路的交换网络。立即发送数据并不会降低数据接受速率。如果没有网络拥塞，向 R 个副本传输 B 字节的数据理想的时间耗费是 B/T+RL，T 代表网络吞吐率，L 是机器间的网络延时。我们的网络连接是 100Mbps(T),L 远远低于 1ms，因此 1MB 的数据理想情况下需要 80ms 就可以完成。

3.3 原子性的记录 append

GFS 提供一个原子性的 append 操作叫做 record append(注意这与传统的 append 操作不同)。在传统的写操作中，用户指定数据需要写的偏移位置，而对于相同区域的并行写操作是不可串行化的：该区域最终可能包含来自多个 client 的数据片段。但在一个 record append 操作中，client 唯一需要说明的只有数据。GFS 会将它至少原子性地 append 到文件中一次，append 的位置是由 GFS 选定的，同时会将这个位置返回给 client。这很类似于 unix 文件打开模式中的 O_APPEND，当多个写者并发操作时不会产生竞争条件。

Record append 在我们的分布式应用中被大量的使用。很多不同机器上的 client 并发地向同一个文件 append。如果使用传统的写操作，client 将需要进行复杂而又昂贵的同步化操作，比如通过一个分布式锁管理器。在我们的工作负载中，这样的文件通常作为一个多生产者/单消费者队列或者用来保存来自多个不同 client 的归并结果。

Record append 是一种类型的变更操作，依然遵循 3.1 节的控制流，只是在主副本上会有一点额外的逻辑。Client 将所有的数据推送给所有副本后，它向主副本发送请求。主副本检查将该记录 append 到该 chunk 是否会导致该 chunk 超过它的最大值(64MB)。如果超过了，它就将该 chunk 填充到最大值，告诉次副本做同样的工作，然后告诉客户端该操作应该在下一个 trunk 上重试。(append 的 Record 大小需要控制在最大 trunk 大小的四分之一以内，这样可以保证最坏情况下的碎片可以保持在一个可以接受的水平上)。如果记录没有超过最大尺寸，就按照普通情况处理，主副本将数据 append 到它的副本上，告诉次副本将数据写在相同的偏移位置上，最后向 client 返回成功应答。

如果 record append 在任何一个副本上失败，client 就会重试这个操作。这样，相同 chunk 的多个副本就可能包含不同的数据，这些数据可能包含了相同记录的整个或者部分的重复值。GFS 并不保证所有的副本在位级别上的一致性，它只保证数据作为一个原子单元最少写入一次。这个属性是由如下的简单观察推导出来的，

当操作报告成功时，数据肯定被写入到某个 trunk 的所有副本的相同偏移位置上{!比如假设数据大小为 200byte，某个副本只写了 100byte 就失败了，当请求再次发起时，主副本会重新写一次，这样数据就应该在偏移为 201 的地方开始，而其他副本必须保证与主副本写在相同的偏移位置，它们就需要再从 201 处开始写，而对于主副本来说它的 100-200 之间就是记录的数据，而失败的副本这块区域根本就没来得及写，所以在这个区域二者是不一致的}。此后，所有的副本至少达到了记录尾部的大小，因此未来的记录将会被放置在更高的偏移位置，或者是另一个不同的 chunk，即使另一个副本变成了主副本。在我们的一致性保证里，record append 操作成功后写下的数据区域是已定义的(肯定是一致的)，然而 介于其间的数据则是不一致的(因此也是未定义的)。我们的应用程序可以处理这样的不一致区域，正如我们在 2.7.2 里讨论的那样。

3.4 快照

快照操作可以非常快速的保存文件或者目录树的一个拷贝，同时可以最小化对于正在执行的变更操作的中断。用户经常用它来创建大数据集的分支拷贝，以及拷贝的拷贝……。或者用来创建检查点，以实验将要提交的拷贝或者回滚到更早的状态。

像 AFS 一样，我们使用标准的写时拷贝技术来实现快照。当 master 收到一个快照请求时，它首先撤销将要进行快照的那些文件对应的 chunk 的所有已发出的租约。这就使得对于这些 chunk 的后续写操作需要与 master 交互来得到租约持有者。这就首先给 master 一个机会创建该 chunk 的新的拷贝。

当这些租约被撤销或者过期后，master 将这些操作以日志形式写入磁盘。然后复制该文件或者目录树的元数据，然后将这些日志记录应用到内存中的复制后的状态上，新创建的快照文件与源文件一样指向相同的 chunk。

当 client 在快照生效后第一次对一个 chunk C 进行写入时，它会发送请求给 master 找到当前租约拥有者。Master 注意到对于 chunk C 的引用计数大于 1。它延迟回复客户端的请求，选择一个新的 chunk handle C`。然后让每个拥有 C 的那些 chunkserver 创建一个新的叫做 C`的 chunk。通过在相同的 chunkserver 上根据原始的 chunk 创建新 chunk，就保证了数据拷贝是本地的，而不是通过网络(我们的硬盘比 100Mb 网络快大概三倍)。这样，对于任何 chunk 的请求处理都没有什么不同：master 为新 chunk C`的副本中的一个授权租约，然后返回给 client，这样它就可以正常的写这个 chunk 了，client 不需要知道该 chunk 实际上是从一个现有的 chunk 创建出来的。

4.master 操作

Master 执行所有的名字空间操作。此外，它还管理整个系统的 chunk 备份：决定如何放置，创建新的 chunk 和相应的副本，协调整个系统的活动保证 chunk 都是

完整备份的，在 chunkserver 间进行负载平衡，回收没有使用的存储空间。我们现在讨论这些主题。

4.1 名字空间管理和锁

很多 master 操作都需要花费很长时间：比如，一个快照操作要撤销该快照所包含的 chunk 的所有租约。我们并不想耽误其他运行中的 master 操作，因此我们允许多个操作同时是活动的，通过在名字空间区域使用锁来保证正确的串行化。

不像传统的文件系统，GFS 的目录并没有一种数据结构用来列出该目录下所有文件，而且也不支持文件或者目录别名(像 unix 的硬链接或者软连接那样)。GFS 在逻辑上通过一个路径全称到元数据映射的查找表来表示它的名字空间。通过采用前缀压缩，这个表可以有效地在内存中表示。名字空间树中的每个节点(要么是文件的绝对路径名称要么是目录的)具有一个相关联的读写锁。

每个 master 操作在它运行前，需要获得一个锁的集合。比如如果它想操作/d1/d2…/dn/leaf，那么它需要获得/d1,/d1/d2……/d1/d2…/dn 这些目录的读锁，然后才能得到路径/d1/d2…/dn/leaf 的读锁或者写锁。Leaf 可能是个文件或者目录，这取决于具体的操作。

我们现在解释一下，当为/home/user 创建快照/save/user 时，锁机制如何防止文件/home/user/foo 被创建。快照操作需要获得在/home 和/save 上的读锁，以及/home/user 和/save/user 上的写锁。文件创建需要获得在/home 和/home/user 上的读锁，以及在/home/user/foo 上的写锁。这两个操作将会被正确的串行化，因为它们试图获取在/home/user 上的相冲突的锁。文件创建并不需要父目录的写锁，因为实际上这里并没有”目录”或者说是类似于 inode 的数据结构，需要防止被修改。读锁已经足够用来防止父目录被删除。

这种锁模式的一个好处就是它允许对相同目录的并发变更操作。比如多个文件的创建可以在相同目录下并发创建：每个获得该目录的一个读锁，以及文件的一个写锁。目录名称上的读锁足够防止目录被删除，重命名或者快照。文件名称上的写锁将会保证重复创建相同名称的文件的操作只会被执行一次。

因为名字空间有很多节点，所以读写锁对象只有在需要时才会被分配，一旦不再使用就删除。为了避免死锁，锁是按照一个一致的全序关系进行获取的：首先根据所处的名字空间树的级别，相同级别的则根据字典序。

4.2 备份放置

GFS 在多个层次上都具有高度的分布式。它拥有数百个散布在多个机柜(rack)中的 chunkserver。这些 chunkserver 又可以被来自不同或者相同机柜上的 client 访问。处在不同机柜的机器间的通信可能需要穿过一个或者更多的网络交换机。此外，进出一个机柜的带宽可能会小于机柜内所有机器的带宽总和。多级的分布式带来

了数据分布式时的扩展性，可靠性和可用性方面的挑战。

Chunk 的备份放置策略服务于两个目的：最大化数据可靠性和可用性，最小化网络带宽的使用。为了达到这两个目的，仅仅将备份放在不同的机器是不够的，这只能应对机器或者硬盘失败，以及最大化利用每台机器的带宽。我们必须在机柜间存放备份。这样能够保证当一个机柜整个损坏或者离线(比如网络交换机故障或者电路出问题)时，该 chunk 的存放在其他机柜的某些副本仍然是可用的。这也意味着对于一个 chunk 的流量，尤其是读取操作可以充分利用多个机柜的带宽。另一方面，写操作需要在多个机柜间进行，但这是我们可以接受的。

4.3 创建 重备份 重平衡(Creation, Re-replication, Rebalancing)

Chunk 副本的创建主要有三个原因：chunk 的创建，重备份，重平衡。

当 master 创建一个 chunk 时，将初始化的空的副本放置在何处，它会考虑几个因素：1.尽量把新的 chunk 放在那些低于平均磁盘空间使用值的那些 chunkserver 上。随着时间的推移，这会使得 chunkserver 的磁盘使用趋于相同 2.尽量限制每个 chunkserver 上的最近的文件创建数，虽然创建操作是很简单的，但是它后面往往跟着繁重的写操作，因为 chunk 的创建通常是因为写者的需要而创建它。在我们的一次 append 多次读的工作负载类型中，一旦写入完成，它们就会变成只读的。3.正如前面讨论的，我们希望在机柜间存放 chunk 的副本

当 chunk 的可用备份数低于用户设定的目标值时，Master 会进行重复制。有多个可能的原因导致它的发生：chunkserver 不可用，chunkserver 报告它的某个备份已被损坏，一块硬盘由于错误而不可用或者用户设定的目标值变大了。需要重复制的 chunk 根据几个因素确定优先级。一个因素是它与备份数的目标值差了多少，比如我们给那些丢失了 2 个副本的 chunk 比丢失了 1 个的更高的优先级。另外，比起最近被删除的文件的 chunk，我们更想备份那些仍然存在的文件的 chunk(参考 4.4 节)。最后，为了最小化失败对于运行中的应用程序的影响，我们提高那些阻塞了用户进度的 chunk 的优先级。

Master 选择最高优先级的 chunk，通过给某个 chunkserver 发送指令告诉它直接从一个现有合法部分中拷贝数据来进行克隆。新备份的放置与创建具有类似的目标：平均磁盘使用，限制在单个 chunkserver 上进行的 clone 操作数，使副本存放在不同机柜间。为了防止 clone 的流量淹没 client 的流量，master 限制整个集群每个 chunkserver 上处在活动状态的 clone 操作数。另外每个 chunkserver 还会限制它用在 clone 操作上的带宽，通过它来控制对源 chunkserver 的读请求。

最后，master 会周期性的对副本进行重平衡。它检查当前的副本分布，然后为了更好地使用磁盘空间和负载平衡，将副本进行移动。而且在这个过程中，master 是逐步填充一个新的 chunkserver，而不是立即将新的 chunk 以及大量沉重的写流量使它忙的不可开交。对于一个新副本的放置，类似于前面的那些讨论。另外，master 必须选择删除哪个现有的副本。通常来说，它更喜欢删除那些存放在低于平均磁盘空闲率的 chunkserver 上的 chunk，这样可以使磁盘使用趋于相等。

4.4 垃圾回收

文件删除后，GFS 并不立即释放可用的物理存储。它会将这项工作推迟到文件和 chunk 级别的垃圾回收时做。我们发现，这种方法使得系统更简单更可靠。

4.4.1 机制

当文件被应用程序删除时，master 会将这个删除操作像其他变化一样立即写入日志。文件不会被立即删除，而是被重命名为一个包含删除时间戳的隐藏名称。在 master 对文件系统进行常规扫描时，它会删除那些存在时间超过 3 天(这个时间是可以配置的)的隐藏文件。在此之前，文件依然可以用那个新的特殊名称进行读取，或者重命名回原来的名称来取消删除。当隐藏文件从名字空间删除后，它的元数据会被擦除。这样就有效地切断了它与所有 chunk 的关联。

在 chunk 的类似的常规扫描中，master 找到那些孤儿(orphaned)块(无法从任何文件到达)，擦除这些块的元数据。在与 master 周期性交互的心跳信息中，chunkserver 报告它所拥有的 chunk 的那个子集，然后 master 返回那些不在 master 的元数据中出现的 chunk 的标识。Chunkserver 就可以自由的删除这些 chunk 的那些副本了。

4.4.2 讨论

尽管程序设计语言中的分布式垃圾回收是一个需要复杂解决方案的难题，但是在我们这里它是很简单的。我们可以简单的找到对于 chunk 的所有引用：因为它们保存在只由 master 维护的一个文件--chunk 映射里。我们可以找到所有 chunk 的副本：它们不过是存放在每个 chunkserver 的特定目录下的 linux 文件。任何 master 不知道的副本就是垃圾。

采用垃圾回收方法收回存储空间与直接删除相比，提供了几个优势：1.在经常出现组件失败的大规模分布式系统中，它是简单而且可靠的。Chunk 创建可能在某些 chunkserver 上成功，在另外一些失败，这样就留下一些 master 所不知道的副本。副本删除消息可能丢失，master 必须记得在出现失败时进行重发。垃圾回收提供了一种统一的可信赖的清除无用副本的方式。2.它将存储空间回收与 master 常规的后台活动结合在一起，比如名字空间扫描，与 chunkserver 的握手。因此它们是绑在一块执行的，这样开销会被平摊。而且只有当 master 相对空闲时才会执行。Master 就可以为那些具有时间敏感性的客户端请求提供更好的响应。3.空间回收的延迟为意外的不可逆转的删除提供了一道保护网。

根据我们的经验，主要的缺点是，当磁盘空间很紧张时，这种延时会妨碍到用户对磁盘使用的调整。那些频繁创建和删除中间文件的应用程序不能够立即重用磁盘空间。我们通过当已删除的文件被再次删除时加速它的存储回收来解决这个问题。我们也允许用户在不同的名字空间内使用不同的重备份和回收策略。比如用户可以指定某个目录树下的文件的 chunk 使用无副本存储，任何已经删除的文件

会被立即删除并且从当前文件系统中彻底删除。

4.5 过期副本检测

如果 chunkserver 失败或者在它停机期间丢失了某些更新，chunk 副本就可能变为过期的。对于每个 chunk，master 维护一个版本号来区分最新和过期的副本。

只要 master 为一个 chunk 授权一个新的租约，那么它的版本号就会增加，然后通知副本进行更新。在一致的状态下，Master 和所有副本都会记录这个新的版本号。这发生在任何 client 被通知以前，因此也就是 client 开始向 chunk 中写数据之前。如果另一个副本当前不可用，它的 chunk 版本号就不会被更新。当 chunkserver 重启或者报告它的 chunk 和对应的版本号的时候，master 会检测该 chunkserver 是否包含过期副本。如果 master 发现有些版本号大于它的记录，master 就认为它在授权租约时失败了，所以采用更高的版本号的那个进行更新。

Master 通过周期性的垃圾回收删除过期副本。在此之前，对于客户端对于该 chunk 的请求 master 会直接将过期副本当作根本不存在进行处理。作为另外一种保护措施，当 master 通知客户端哪个 chunkserver 包含某 chunk 的租约或者当它在 clone 操作中让 chunkserver 从另一个 chunkserver 中读取 chunk 时，会将 chunk 的版本号包含在内。当 clinet 和 chunkserver 执行操作时，总是会验证版本号，这样就使得它们总是访问最新的数据。

5.容错和诊断

在设计系统时，一个最大的挑战就是频繁的组件失败。组件的数量和质量使得这些问题变成一种常态而不再是异常。我们不能完全信任机器也不能完全信任磁盘。组件失败会导致系统不可用，甚至是损坏数据。我们讨论下如何面对这些挑战，以及当它们不可避免的发生时，在系统中建立起哪些工具来诊断问题。

5.1 高可用性

在 GFS 的数百台服务器中，在任何时间总是有一些是不可用的。我们通过两个简单有效的策略来保持整个系统的高可用性：快速恢复和备份。

5.1.1 快速恢复

Master 和 chunkserver 都设计得无论怎么样地被终止，都可以在在几秒内恢复它们的状态并启动。事实上，我们并没有区分正常和异常的终止。服务器通常都是通过杀死进程来关闭。客户端和其他服务器的请求超时后会经历一个小的停顿，然后重连那个重启后的服务器，进行重试。6.2.2 报告了观测到的启动时间。

5.1.2chunk 备份

正如之前讨论的，每个 chunk 备份在不同机柜上的多个 chunkserver 上。用户可以在不同名字空间内设置不同的备份级别，默认是 3。当 chunkserver 离线或者通过检验检测到某个 chunk 损坏后(5.2 节)，master 会克隆现有的副本使得副本的数保持充足。尽管副本已经很好的满足了我们的需求，我们还在探寻一些其他的跨机器的冗余方案(比如使用 parity or erasure 编码){! parity code，奇偶校验码。erasure code，将一个数据文件划分成等长的 n 个数据块(不足以 0 补充)，通过编码生成 m 个校验块，在这 n+M 个块中，只要有其中至少 n 个，就可以恢复出原始数据，这样就能容忍多达 m 个节点的失效}，来满足我们日益增长的只读存储需求。我们期望在我们的非常松散耦合的系统中实现这些更复杂的冗余模式是具有挑战性但是可管理的。因为我们的负载主要是 append 和读操作而不是小的随机写操作。

5.1.3master 备份

为了可靠性，master 的状态需要进行备份。它的操作日志和检查点备份在多台机器上。对于状态的变更只有当它的操作日志被写入到本地磁盘和所有的远程备份后，才认为它完成。为了简单起见，master 除了负责进行各种后台活动比如：垃圾回收外，还要负责处理所有的变更。当它失败后，几乎可以立即重启。如果它所在的机器或者硬盘坏了，独立于 GFS 的监控设施会利用备份的操作日志在别处重启一个新的 master 进程。Client 仅仅使用 master 的一个典型名称(比如 gfs-test)来访问它，这是一个 DNS 名称，如果 master 被重新部署到一个新的机器上，可以改变它。

此外，当主 master down 掉之后，还有多个影子(shadow)master 可以提供对文件系统的只读访问。它们是影子，而不是镜像，这意味着它们可能比主 master 要滞后一些，通常可能是几秒。对于那些很少发生变更的文件或者不在意轻微过时的应用程序来说，它们增强了读操作的可用性。实际上，因为文件内容是从 chunkserver 中读取的，应用程序并不会看到过期的文件内容。文件元数据可能在短期内是过期的，比如目录内容或者访问控制信息。

为了保持自己的实时性，影子服务器会读取不断增长的操作日志的副本，然后像主 master 那样将这些变化序列应用在自己的数据结构上。与主 master 一样，它也会在启动时向 chunkserver 拉数据来定位 chunk 的副本，也会同它们交换握手信息以监控它们的状态。只有在主 master 决定创建或者删除副本时引起副本位置信息更新时，它才依赖于主 master。

5.2 数据完整性(Data Integrity)

每个 chunkserver 通过校验和(checksumming)来检测存储数据中的损坏。GFS 集群通常具有分布在几百台机器上的数千块硬盘，这样它就会经常出现导致数据损坏

或丢失的硬盘失败。我们可以从 chunk 的其他副本中恢复被损坏的数据，但是如果通过在 chunkserver 间比较数据来检测数据损坏是不现实的。另外，有分歧的备份仍然可能是合法的：根据 GFS 的变更语义，尤其是前面提到的原子性的 record append 操作，并不保证所有副本是完全一致的。因此每个 chunkserver 必须通过维护一个检验和来独立的验证它自己的拷贝的完整性(integrity)。

一个 chunk 被划分为 64kb 大小的块。每个块有一个相应的 32bit 的校验和。与其他的元数据一样，校验和与用户数据分离，它被存放在内存中，同时通过日志进行持久化存储。

对于读操作，chunkserver 在向请求者(可能是一个 client 或者其他的 chunkserver)返回数据前，需要检验与读取边界重叠的那些数据块的校验和。因此 chunkserver 不会将损坏数据传播到其他机器上去。如果一个块的校验和与记录中的不一致，chunkserver 会向请求者返回一个错误，同时向 master 报告这个不匹配。之后，请求者会向其他副本读取数据，而 master 则会用其他副本来 clone 这个 chunk。当这个合法的新副本创建成功后，master 向报告不匹配的那个 chunkserver 发送指令删除它的副本。

校验和对于读性能的影响很小，因为：我们大部分的读操作至少跨越多个块，我们只需要读取相对少{!与需要读取的所有数据相比}的额外数据来进行验证。GFS client 代码通过尽量在校验边界上对齐读操作大大降低了开销。另外在 chunkserver 上校验和的查找和比较不需要任何的 IO 操作，校验和的计算也可以与 IO 操作重叠进行。

校验和计算对于 append 文件末尾的写操作进行了特别的优化。因为它们在工作负载中占据了统治地位。我们仅仅增量性的更新最后一个校验块的校验值，同时为那些 append 尾部的全新的校验块计算它的校验值。即使最后一个部分的校验块已经损坏，而我们现在无法检测出它，那么新计算出来的校验和将不会与存储数据匹配，那么当这个块下次被读取时，就可以检测到这个损坏。{!也就是说这里并没有验证最后一个块的校验值，而只是更新它的值，也就是说这里省去了验证的过程，举个例子假设之后最后一个校验块出现了错误，由于我们的校验值计算是增量性的，也就是说下次计算不会重新计算已存在的这部分数据的校验和，这样该损坏就继续保留在校验和里，关键是因为这里采用了增量型的校验和计算方式}

与之相对的，如果一个写操作者覆盖了一个现有 chunk 的边界，我们必须首先读取和验证操作边界上的第一个和最后一个块，然后执行写操作，最后计算和记录新的校验和。如果在覆盖它们之前不验证第一个和最后一个块，新的校验和就可能隐藏掉那些未被覆盖的区域的数据损坏。{!因为这里没有采用增量计算方式，因为它是覆盖不是 append 所以现有的检验和就是整个块的没法从中取出部分数据的校验和，必须重新计算}

在空闲期间，chunkserver 可以扫描验正处在非活动状态的 trunk 的内容。这允许我们检测到那些很少被读取的数据的损坏。一旦损坏被发现，master 就可以创建

一个新的未损坏副本并且删除损坏的副本。这就避免了一个不活跃的坏块骗过master，让之以为该块有足够的好的副本。

5.3 诊断工具

全面详细的诊断性日志以很小的成本，带来了在问题分解、调试、性能分析上不可估量的帮助。没有日志，就很难理解那些机器间偶然出现的不可重现的交互。GFS 生成一个诊断日志用来记录很多重要事件(比如 chunkserver 的启动停止)以及所有 RPC 请求和应答。这些诊断日志可以自由的删除而不影响系统的正常运行。然而，只要磁盘空间允许，我们会尽量保存这些日志。

RPC 日志包含了所有的请求和响应信息，除了读写的文件数据。通过匹配请求和响应，整理不同机器上的 RPC 日志，我们可以重新构建出整个交互历史来诊断一个问题。这些日志也可以用来进行负载测试和性能分析。

因为日志是顺序异步写的，因此写日志对于性能的影响是很小的，得到的好处却是大大的。最近的事件也会保存在内存中，可以用于持续的在线监控。

6.测量(MEASUREMENTS)

在这一节，我们用一些小规模的测试来展示 GFS 架构和实现固有的一些瓶颈，有一些数字来源于 google 实际使用中的集群。

6.1 小规模测试

我们在一个由一个 master，两个 master 备份，16 个 chunkserver，16 个 client 组成的 GFS 集群上进行了性能测量。这个配置是为了方便测试，实际中的集群通常会有数百个 chunkserver，数百个 client。

所有机器的配置是，双核 PIII 1.4GHz 处理器，2GB 内存，两个 80G、5400rpm 硬盘，以及 100Mbps 全双工以太网连接到 HP2524 交换机。所有 19 个 GFS 服务器连接在一个交换机，所有 16 个客户端连接在另一个上。两个交换机用 1Gbps 的线路连接。

6.1.1 读操作

N 个客户端从文件系统中并发读。每个客户端在一个 320GB 的文件集合里随机选择 4MB 的区间进行读取。然后重复 256 次，这样每个客户端实际上读取了 1GB 数据。Chunkserver 总共只有 32GB 内存，因此我们估计在 linux 的 buffer cache 里最多有 10%的命中率。我们的结果应该很接近一个几乎无缓存的结果。

图 3(a)展示了对于 N 个客户端的总的读取速率以及它的理论上的极限。当 2 个交换机通过一个 1Gbps{!b 是 bit，根据惯例，GBps 中的 B 才是字节}的链路连接时，它的极限峰值是 125MB/s，客户端通过 100Mbps 连接，那么换成单个客户端的极限就是 12.5MB/s。当只有一个客户端在读取时，观察到的读取速率是 10MB/s，达到了单个客户端极限的 80%。当 16 个读取者时，总的读取速率的 94 MB/s，大概达到了链路极限(125MB/s)的 75%，换成单个客户端就是 6 MB/s。效率从 80%降到了 75%，是因为伴随着读取者的增加，多个读者从同一个 chunkserver 并发读数据的概率也随之变大。

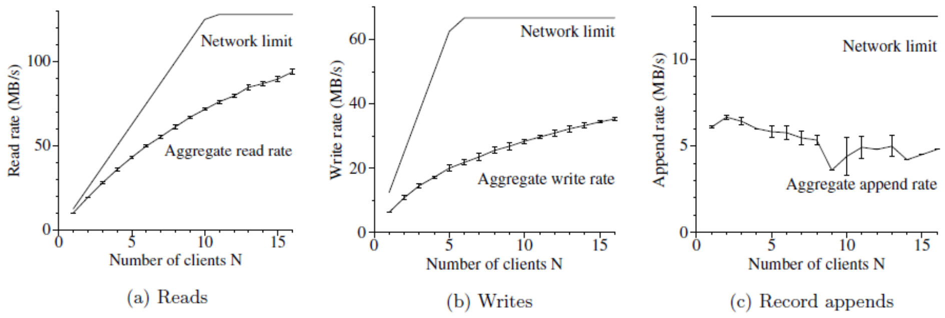


Figure 3: Aggregate Throughputs. Top curves show theoretical limits imposed by our network topology. Bottom curves show measured throughputs. They have error bars that show 95% confidence intervals, which are illegible in some cases because of low variance in measurements.

6.1.2 写操作

N 个客户端并行向 N 个不同的文件写数据。每个客户端以 1MB 的单个写操作总共向一个新文件写入 1GB 数据。总的写速率以及它的理论上的极限如图 3(b)所示。极限值变成了 67 MB/s，是因为我们需要将每个字节写入到 16 个 chunkserver 中的 3 个，每个具有 12.5MB/s 的输入连接。

单个客户端的写入速率是 6.3 MB/s，大概是极限值的一半。主要原因是我们的网络协议栈。它不能充分利用我们用于 chunk 副本数据推送的流水线模式。将数据从一个副本传递到另一个副本的延迟降低了整体的写速率。

对于 16 个客户端，总体的写入速率达到了 35 MB/s，平均每个客户端 2.2 MB/s,大概是理论极限的一半。与写操作类似，伴随着写者的增加，多个写者从同一个 chunkserver 并发写数据的概率也随之变大。另外对于 16 个写者比 16 个读者更容易产生碰撞，因为每个写者将关联到 3 个不同的副本。

写者比我们期望的要慢。在实际中，这还未变成一个主要问题，因为尽管它可能增加单个客户端的延时，但是当系统面对大量客户端时，其总的写入带宽并没有显著的影响。

6.1.3 记录追加

图 3(c)展示了 record append 的性能。N 个客户端向单个文件并行的 append。性

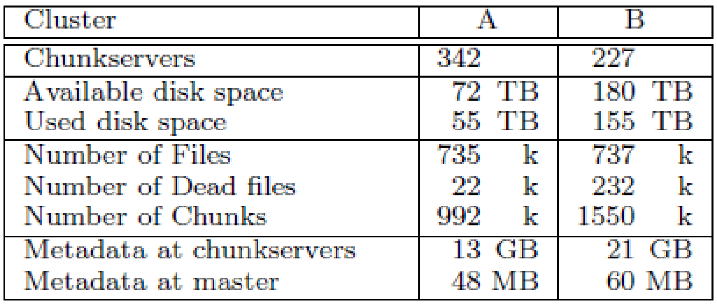
能取决于保存了该文件最后那个 chunk 的那些 chunkserver，与客户端的数目无关。当只有一个客户端时，能达到6.0MB/s，当有16个客户端时就降到了4.8 MB/s。主要是由于拥塞以及不同的客户端的网络传输速率造成的。

我们的应用程序倾向于并行创建多个这样的文件。换句话说，N 个客户端向 M 个共享文件并行 append，在这里 N 和 M 通常是几十甚至几百大小。因此在我们的实验中出现的 chunkserver 的网络拥塞问题在实际中并不是一个显著的问题，因为当一个文件的 chunkserver 比较繁忙的时候，它可以去写另一个。

6.2 现实中的集群

我们选择在 google 内部使用的两个集群进行测试作为相似的那些集群的一个代表。集群 A 主要用于 100 多个工程的日常研发。它会从数 TB 的数据中读取数 MB 的数据，对这些数据进行转化或者分析，然后将结果再写回集群。集群 B 主要用于产品数据处理。它上面的任务持续时间更长，持续地在生成和处理数 TB 的数据集合，只是偶尔可能需要人为的参与。在这两种情况下，任务都是由分布在多个机器上的很进程组成，它们并行地读写很多文件。

6.2.1 存储



Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

Table 2: Characteristics of two GFS clusters

正如表中前 5 个字段所展示的，两个集群都有数百个 chunkserver，支持 TB 级的硬盘空间，空间已经被充分使用但还没全满。已用的空间包含 chunk 的所有副本。通常文件存在三个副本，因此这两个集群实际分别存储了 18TB 和 52TB 的数据。

这两个集群的文件数目很接近，尽管 B 集群有大量的死文件(那些已经被删除或者被新版本文件所替换但空间还没有被释放的文件)。而且它具有更多的 trunk，因为它上面的文件通常更大。

6.2.2 元数据

所有的 Chunkserver 总共存储了数十 G 的元数据，大部分是用户数据的 64KB 块的校验和。Chunkserver 上唯一的其他的元数据就是 4.5 节讨论的 chunk 的版本号。

保存在 master 上的元数据要更小一些，只有数十 MB，平均下来每个文件只有 100 来个字节。这也刚好符合我们的 master 的内存不会成为实际中系统容量限制的料想。每个文件的元数据主要是以前缀压缩格式存储的文件名称。还有一些其他的元数据比如文件所有者，权限，文件到 chunk 的映射以及 chunk 的当前版本。另外对于每个 chunk 我们还存储了当前的副本位置以及用于实现写时复制的引用计数。

每个独立的 server(chunkserver 和 master)只有 50-100MB 的元数据。因此，恢复是很快的：在 server 可以应答查询前只需要花几秒钟的时间就可以把它们从硬盘上读出来。然而，master 的启动可能要慢一些，通常还需要 30-60 秒从所有的 chunkserver 获得 chunk 的位置信息。

6.2.3 读写速率

表 3 展示了不同时期的读写速率。进行这些测量时，两个集群都已经运行了大约一周(为了更新到最新版本的 GFS，这两个集群被重启过)。

Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

Table 3: Performance Metrics for Two GFS Clusters

从启动开始看，平均写速率小于 30MB/s。当我们进行这些测量时，集群 B 正在以 100MB/s 的速率进行密集的写操作，同时产生了 300MB/s 的网络负载，因为写操作将会传给 3 个副本。

读速率要远高于写速率。正如我们料想的那样，整个工作负载组成中，读要多于写。这两个集群都处在繁重的读活动中。尤其是，A 已经在过去的一个星期中维持了 580MB/s 的读速率。它的网络配置可以支持 750MB/s，因此它已经充分利用了资源。B 集群可支持 1300 MB/s 的峰值读速率，但是应用只使用了 380 MB/s。

6.2.4 master 负载

表 3 也表明发送给 master 的操作速率大概是每秒 200-500 个操作。Master 可以

轻易的处理这个级别的速率，因此对于这些工作负载来说，它不会成为瓶颈。

在早期版本的 GFS 中，master 偶尔会成为某些工作负载的瓶颈。为了查找文件，花费大量的时间在巨大的目录(包含上千万的文件)中进行线性扫描。因此，我们改变了 master 的数据结构，使之可以在名字空间内进行有效的二分搜索。现在它可以简单的支持每秒上千次的文件访问。如果必要的话，我们还可以进一步的在名字空间数据结构前端提供名字查找缓存。

6.2.5 恢复时间

一台 Chunkserver 失败后，它上面的那些 chunk 的副本数就会降低，必须进行 clone 以维持正常的副本数。恢复这些 chunk 的时间取决于资源的数量。在一个实验中，我们关闭集群 B 中的一个 chunkserver。该 chunkserver 大概有 15000 个 chunk，总共 600GB 的数据。为减少对于应用程序的影响以及为调度决策提供余地，我们的默认参数设置将集群的并发 clone 操作限制在 91 个(占 chunkserver 个数的 40%),同时每个 clone 操作最多可以消耗 6.25MB/s(50Mbps)。所有的 chunk 在 23.2 分钟内被恢复，备份速率是 440MB/s。

在另一个实验中，我们关掉了两个 chunkserver，每个具有 16000 个 chunk，660GB 的数据。这次失败使得 266 个 chunk 降低到了一个副本，但是两分钟内，它们就恢复到了至少 2 个副本，这样就让集群能够容忍另一个 chunkserver 发生失败，而不产生数据丢失。

6.3 工作负载剖析

在这一节，我们将继续在两个新的集群上对工作负载进行细致的对比分析。集群 X 用于研究开发，集群 Y 用于产品数据处理。

6.3.1 方法和说明

这些结果只包含了客户端产生的请求，因此它们反映了应用程序对整个文件系统的工作负载。并不包含为了执行客户端的请求进行的 server 间的请求，或者是内部的后台活动，比如写推送或者是重平衡。

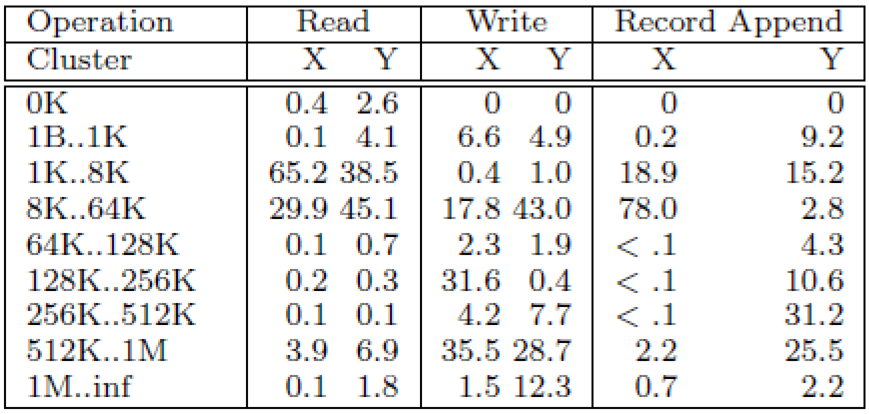
对于 IO 操作的统计是从 GFS 的 server 的 PRC 请求日志中重新构建出来的。比如为了增加并行性，GFS 客户端代码可能将一个读操作拆分为多个 RPC 请求，我们通过它们推断出原始请求。因为我们的访问模式高度的程式化，希望每个错误都可以出现在日志中。应用程序显式的记录可以提供更精确的数据，但是重新编译以及重启正在运行中的客户端在逻辑上是不可能这样做的。而且由于机器数很多，收集这些数据也会变得很笨重。

需要注意的是，不能将我们的工作负载过于一般化。因为 GFS 和应用程序是由 google 完全控制的，应用程序都是针对 GFS 进行专门优化的，同时 GFS 也是专门

为这些应用而设计的。这种相互的影响可能也存在于一般的文件系统及其应用程序中，但是这种影响可能并不像我们上面所描述的那样。

6.3.2 chunkserver 负载

表 4 展示了操作根据大小的分布。读操作的大小表现出双峰分布，小型读操作(小于 64KB)来自于那些在大量文件中查找小片数据的随机读客户端，大型读操作(超过 512KB)来自于穿越整个文件的线性读操作。



Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
0K	0.4	2.6	0	0	0	0
1B..1K	0.1	4.1	6.6	4.9	0.2	9.2
1K..8K	65.2	38.5	0.4	1.0	18.9	15.2
8K..64K	29.9	45.1	17.8	43.0	78.0	2.8
64K..128K	0.1	0.7	2.3	1.9	< .1	4.3
128K..256K	0.2	0.3	31.6	0.4	< .1	10.6
256K..512K	0.1	0.1	4.2	7.7	< .1	31.2
512K..1M	3.9	6.9	35.5	28.7	2.2	25.5
1M..inf	0.1	1.8	1.5	12.3	0.7	2.2

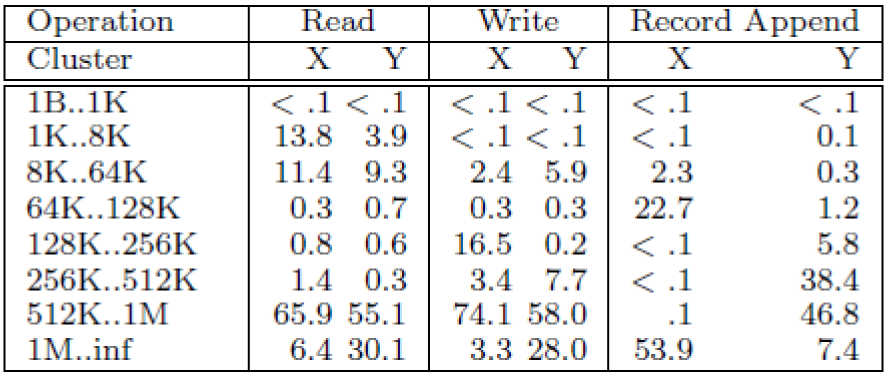
Table 4: Operations Breakdown by Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested.

集群 Y 中大量的读操作没有返回数据。我们应用程序，尤其是在产品系统中，经常使用文件作为生产者消费者队列。生产者并行的往文件中 append 数据，而消费者则从文件尾部读数据。有时候，如果消费者超过了生产者，就没有数据返回。集群 X 很少出现这种情况，因为它主要是用来进行短期数据分析，而不是长期的分布式应用。

写操作的大小也表现出双峰分布。大型的写操作(超过 256KB)通常来自于写操作者的缓冲。那些缓冲更少数据的写操作者，检查点或者经常性的同步或者简单的数据生成组成了小型的写操作(低于 64KB)。

对于记录的 append，Y 集群比 X 集群可以看到更大的大 record append 比率。因为使用 Y 集群的产品系统，针对 GFS 进行了更多的优化。

表 5 展示了不同大小的数据传输总量。对于各种操作来说，大型的操作(超过 256KB)构成了大部分的数据传输。但是小型(低于 64KB)的读操作虽然传输了比较少的数据但是在数据读中也占据了相当的一部分，主要是由于随机 seek 造成的。



Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
1B..1K	< .1	< .1	< .1	< .1	< .1	< .1
1K..8K	13.8	3.9	< .1	< .1	< .1	0.1
8K..64K	11.4	9.3	2.4	5.9	2.3	0.3
64K..128K	0.3	0.7	0.3	0.3	22.7	1.2
128K..256K	0.8	0.6	16.5	0.2	< .1	5.8
256K..512K	1.4	0.3	3.4	7.7	< .1	38.4
512K..1M	65.9	55.1	74.1	58.0	.1	46.8
1M..inf	6.4	30.1	3.3	28.0	53.9	7.4

Table 5: Bytes Transferred Breakdown by Operation Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested. The two may differ if the read attempts to read beyond end of file, which by design is not uncommon in our workloads.

6.3.4 append 与 write

记录 append 操作被大量的应用，尤其是在我们的产品系统中。对于集群 X 来说，按字节传输来算，write 与 append 的比例是 108：1，根据操作数来算它们的比例是 8：1。对于集群 Y，比例变成了 3.7：1 和 2.5：1。对于这两个集群来说，它们的 append 操作都要比 write 操作大一些{!操作数的比要远大于字节数的比，说明单个的 append 操作的字节数要大于 write}。对于集群 X 来说，在测量期间的记录 append 操作要低一些，这可能是由其中具有特殊缓冲大小设置的应用程序造成的。

正如期望的，我们的数据变更操作处于支配地位的是追加而不是重写{write 也可能是追加}。我们测量了在主副本上的数据重写数量。对于集群 X 来说，以字节大小计算的话重写大概占了整个数据变更的 0.0001%，以操作个数计算，大概小于 0.0003%。对于 Y 集群来说，这两个数字都是 0.05%，尽管这也不算大，但是还是要高于我们的期望。结果显示，大部分的重写是由于错误或者超时导致的客户端重写而产生的。它们并不是工作负载的一部分，而是属于重试机制。

6.3.4 master 负载

表 6 展示了对于 master 各种请求类型的剖析。大部分请求是为了得到 chunk 位置以及数据变更需要的租约持有信息。

可以看到集群 X 和 Y 在 delete 请求上的限制区别，因为集群 Y 上存储的产品信息会被周期性地生成的新版本数据所替换。这些不同被隐藏在 open 请求中，因为老版的数据在被写的时候的打开操作中被隐式的删除(类似与 Unix 的”w”打开模

式)。

查找匹配文件是一个类似于 ls 的模式匹配请求。不像其他的请求，它可能需要处理很大部分的名字空间，因此可能是很昂贵的。在集群 Y 上可以更频繁地看到它，因为自动化的数据处理任务为了了解整个应用程序的状态可能需要检查文件系统中的某些部分。与此相比，集群 X 需要更多显式的用户控制而且已经提前知道所需要的文件的名称。

Cluster	X	Y
Open	26.1	16.3
Delete	0.7	1.5
FindLocation	64.3	65.8
FindLeaseHolder	7.8	13.4
FindMatchingFiles	0.6	2.2
All other combined	0.5	0.8

Table 6: Master Requests Breakdown by Type (%)

7.经验

在构建和部署 GFS 的过程中，我们总结出了很多经验，观点和技术。

起初，GFS 只是考虑作为我们产品系统的后端文件系统。随着时间的推移，开始在研究和开发中使用。一开始它基本不支持像权限，quota 这些东西，但是现在它们都已经有了。产品系统是很容易控制的，但是用户却不是。因此需要更多的设施来避免用户间的干扰。

我们最大的问题是硬盘和 linux 相关性。我们的很多硬盘声称支持各种 IDE 协议版本的 linux 驱动，但是实际上它们只能在最近的一些上才能可靠的工作。因此如果协议版本如果相差不大，硬盘大多数情况下都可以工作，但是有时候这种不一致会使得驱动和内核在硬盘状态上产生分歧。由于内核的问题，这将会导致数据被默默的污染。这个问题使得我们使用校验和来检测数据污染，如果出现这种情况，我们就需要修改内核来处理这种协议不一致的情况。

之前，由于 linux2.2 内核的 fsync()的开销，我们也碰到过一些问题。它的开销是与文件大小而不是被修改部分的大小相关的。这对于我们大的操作日志会是一个问题，尤其是在我们实现检查点之前。我们通过改用同步写来绕过了这个问题，最后迁移到 Linux2.4 来解决了它。

另一个由于 linux 产生的问题是与读写锁相关的。在一个地址空间里的线程在从硬盘中读页数据(读锁)或者在 mmap 调用中修改地址空间(写锁)的时候，必须持有一个读写锁。在系统负载很高，产生资源瓶颈或者出现硬件失败时，我们碰到了瞬态的超时。最后，我们发现当磁盘读写线程处理前面映射的数据时，这个锁

阻塞了网络线程将新的数据映射到内存。由于我们的工作瓶颈主要是在网络带宽而不是内存带宽，因此我们通过使用 pread()加上额外的开销替代 mmap()绕过了这个问题。

尽管出现了一些问题，linux 代码的可用性帮助了我们探索和理解系统的行为。在适当的时机，我们也会改进内核并与开源社区共享这些变化。

8.相关工作

像其他的大型分布式文件系统比如 AFS，GFS 提供了一个本地的独立名字空间，使得数据可以为了容错或者负载平衡而透明的移动。但与 AFS 不同的是，为了提升整体的性能和容错能力，GFS 将文件数据在多个存储服务器上存储，这点更类似于 xFS 或者 Swift。

硬盘是相对便宜的，而且与复杂的 RAID 策略相比，副本策略更简单。由于 GFS 完全采用副本策略进行冗余因此它会比 xFS 或者 Swift 消耗更多的原始存储。
与 AFS,xFS,Frangipani,Intermezzo 这些系统相比，GFS 在文件系统接口下并不提供任何缓存。我们的目标工作负载类型对于通常的单应用程序运行模式来说，基本上是不可重用的，因为这种模式通常需要读取大量数据集合或者在里面进行随机的 seek 而每次只读少量的数据。

一些分布式文件系统比如 xFS,Frangipani,Minnesota’s GFS 和 GPFS 删除了中央服务节点，依赖于分布式的算法进行一致性和管理。我们选择中央化策略是为了简化设计，增加可靠性，获取灵活性。尤其是，一个中央化的 master 更容易实现复杂的 chunk 放置和备份策略，因为 master 具有大部分的相关信息以及控制了它们的改变。我们通过让 master 具有最小化的状态以及在其他机器上进行备份来解决容错。当前通过影子 master 机制提供可扩展性和可用性。对于 master 状态的更新，通过 append 到 write-ahead 日志里进行持久化。因此我们可以通过类似于 Harp 里的主 copy 模式来提供一个比我们当前模式具有更强一致性的高可用性。

我们正在解决类似于 Lustre 的一个问题：提高大量客户端访问时的整体性能。通过专注于我们自己的需求而不是构建一个 POSIX 兼容文件系统，大大简化了这个问题。另外，GFS 假设不可靠组件的数量是很大的，因此容错性是我们设计的核心。

GFS 非常类似于 NASD 架构。但是 NASD 是基于网络连接的硬盘驱动器，GFS 则使用普通机器作为 chunkserver。与 NASD 不同，chunkserver 在需要时分配固定大小的 chunk，而没有使用变长对象。此外，GFS 还实现了诸如重平衡，副本，产品环境需要的快速恢复。

不像 Minnesota’s GFS 和 NASD，我们并没有寻求改变存储设备的模型。我们更专注于解决使用现有商品化组件组成的复杂分布式系统的日常数据处理需求。

通过在生产者消费者队列中使用原子 record append 操作解决了与分布式操作系统 River 的类似问题。River 使用基于内存的跨机器分布式队列以及小心的数据流控制来解决这个问题，而 GFS 只使用了一个可以被很多生产者 append 数据的文件。River 模型支持 m to n 的分布式队列，但是缺乏容错，GFS 目前只支持 m to 1。多个消费者可以读取相同文件，但是它们必须协调好对输入负载进行划分(各自处理不相交的一部分)。

9.总结

GFS 包含了那些在商品化硬件上支持大规模数据处理的必要特征。尽管某些设计决定与我们特殊的应用类型相关，但是可以应用在具有类似需求和特征的数据处理任务中。

针对我们当前的应用负载类型，我们重新审视传统的文件系统的一些假设。这使得我们的设计中产生了一些与之根本不同的观点。我们将组件失败看做常态而不是异常，为经常进行的在大文件上的 append 进行优化，然后是读(通常是顺序的)，为了改进整个系统我们扩展并且放松了标准文件系统接口。

我们的系统通过监控、备份关键数据、快速和自动恢复来提供容错。Chunk 备份使得我们可以容忍 chunkserver 的失败。这些经常性的失败，驱动了一个优雅的在线修复机制的产生，它周期性地透明地进行修复，尽快地恢复那些丢失的副本。另外，我们通过使用校验和来检测数据损坏，当系统中硬盘数目很大的时候，这种损坏变得很正常。

我们的设计实现了对于很多执行大量任务的并发读者和写者的高吞吐率。我们通过从数据传输中分离文件系统控制，来实现这个目标，让 master 来处理文件系统控制，数据传输则直接在 chunkserver 和客户端之间进行。通过增大 chunk 的大小以及 chunk 的租约机制，降低了 master 在普通操作中的参与。这使中央的 master 不会成为瓶颈。我们相信在当前网络协议栈上的改进将会突破目前单个客户端的写吞吐率的限制。

GFS 成功地满足了我们的存储需求，同时除了作为产品数据处理平台外，还作为研发的存储平台而被广泛使用。它是一个使我们可以持续创新以及面对整个 web 的海量数据挑战的重要工具。

致谢

…….

译考文献

1. http://www.cppblog.com/jack-wang/archive/2010/02/26/108503.html 谷歌技术“三宝”之一的Google文件系统和Kosmos 文件系统
2. http://tech.ddvip.com/2009-02/1235810563110017.html分布式基础学习【一】分布式文件系统
3. http://blog.s135.com/book/gfs/ Google文件系统论文

MapReduce:简化大规模集群上的数据处理

作者：Jeffrey Dean & Sanjay Ghemawat Google Inc 2004
译者：phylips@bmy 2010-10-2
出处：http://duanple.blog.163.com/blog/static/709717672010923203501/

摘要

MapReduce 是一个编程模型以及用来处理和生成大数据集的一个相关实现。用户通过描述一个 map 函数，处理一组 key/value 对进而生成一组 key/value 对的中间结果，然后描述一个 reduce 函数，将具有相同 key 的中间结果进行归并。如文中所述，很多现实世界中的任务都可以用这个模型来表达。

以这种函数式风格写出来的程序会在一个由普通机器组成的集群上被自动的进行并行化和执行。由一个运行时系统来关注输入数据的划分细节、在机器集合上的程序执行调度、处理机器失败以及管理所需要的机器间的通信。这就允许那些没有并行分布式系统编程经验的程序员很容易的使用大型分布式系统的资源。

我们的 MapReduce 实现运行在一个由很多普通机器组成的集群上，而且具有高扩展性：一个典型的 MapReduce 计算将会在一个数千台机器的集群上处理 TB 级的数据。对于程序员来说，这个系统很好用，目前已经有数百个 MapReduce 程序实现，在 google 的集群上每天有上千个 MapReduce job 在跑。

1.导引

在过去的五年来，作者和 google 的其他工程师已经实现了数百个用于特殊目的的在大量原始数据(比如爬虫爬的文档，web 访问日志等等)上进行的运算。为了计算各种类型的衍生数据：比如倒排索引，网页文档各种不同的图结构表示，每个 host 的网页数，给定一天中的最常查询集合。大部分这样的计算在概念上都是很简单的。然而由于输入数据通常是很庞大的，因此为了能在合理的时间内结束，计算就不得不分布在成百上千台机器上执行。如何并行化计算、分布数据、处理错误都会使得原本简单的计算需要大量额外的代码去处理这些问题。

为了应对这种复杂性，我们设计了一种抽象，使得我们可以表达我们需要执行的这种简单的运算，而将并行化、容错、数据分布、负载平衡这样的细节封装在库里。我们的抽象源于 Lisp 以及其他函数式编程语言中的 map-reduce 原语。我们发现我们大部分的计算都是首先在输入的每条记录上执行一个 map 操作以产生一个 key/value 的中间结果集合，然后为了得到相应的派生数据，对那些具有相同 key 的值应用一个 reduce 操作。通过使用由用户描述的 map 和 reduce 操作组成的函数式模型，使得我们很容易的进行计算并行化，同时使用重执行

(re-execution)作为基本的容错机制。

这项工作的主要贡献就是提供了一个允许对大规模计算进行自动并行化以及数据分布的简单有力的接口。同时提供了一个可以在普通 pc 机组成的大规模集群上达到很高性能的针对该接口的实现。

第 2 节描述了基本的编程模型并且给出了几个简单例子。第 3 节描述了一个面向基于集群的计算环境的该接口的实现。第 4 节描述了该模型中我们认为很有用的几个概念。第 5 节对我们的实现通过几个 task 进行了测试。第 6 节介绍了 MapReduce 在 google 内部的使用，包括使用它重写我们的索引系统的一些经验。第 7 节讨论了相关的以及未来的工作。

2.编程模型

计算有一个 key/value 输入对集合，产生一系列的输出 key/value 对。MapReduce 库的用户通过两个函数：Map 和 Reduce 来表达这个计算。

Map，由用户编写，有一个输入对，产生一集 key/value 对的中间结果。MapReduce 库将具有相同 key 值的那些中间值组织起来，然后将它们传给 Reduce 函数。

Reduce 函数，也是由用户编写，接受一个中间 key 值，以及对应于该 key 的 value 集合作为输入。它将这些 value 归并起来形成一个可能更小的 value 集合。通常每个 Reduce 调用产生 0 个或者 1 个输出值。中间值的 value 集合是通过一个迭代器来提供给用户的 Reduce 函数。这允许我们能处理那些太大以至于无法一次放入内存的 value 列表。

2.1 例子

考虑在一个大量文档集合中计算单词出现频率的问题。用户可以用类似如下伪代码的方式来编写代码。

```
map(String key, String value):
// key: document name
// value: document contents
for each word w in value:
EmitIntermediate(w, "1");

reduce(String key, Iterator values):
// key: a word
// values: a list of counts
int result = 0;
for each v in values:
result += ParseInt(v);
```

```
Emit(AsString(result));
```
Map 函数输出每个单词及其相应出现次数(在这个简单例子中，就是 1) Reduce 函数将所有的次数加起来然后为每一个单词输出它。

另外用户还需要向一个 MapReduce 描述对象中填写输入输出文件名称以及一些可选的参数。然后用户调用 MapReduce 函数，将该描述对象传给它。用户代码需要链接 MapReduce 库(采用 c++实现)。附录 A 包含该例子的完整代码。

2.2 类型

尽管前面的例子是以字符串类型作为输入输出，概念上来说用户提供的 map 和 reduce 函数有如下的类型关联：
```
Map:     (k1,v1)        -> list(k2,v2)
Reduce:  (k2,list(v2))  -> list(v2)
```
即{!准确的说应该是 Map 端的}输入的 key 和 value 与输出的 key 和 value 来自于不同的域，另外中间结果的 key value 与 Reduce 端的 key value 具有相同的域。{!->前后分别代表了输入输出，所以 list(k2,v2)代表了中间结果，可以看到中间结果与 Reduce 的 key value 名称都是 k2,v2,以表示它们具有相同的域，以 C++的角度看就是具有相同的类型。而 k1 和 k2，v1 和 v2 所代表的实际含义可能是不同的，比如 url 访问频率计数,map 的输入就是<logname,log>,而中间结果与 reduce 则都是<url,出现次数>}

我们的 C++实现向用户传递或者接收来自用户的字符串格式内容，将字符串与相应类型的转换交给用户代码处理。

2.3 更多的实例

下面有一些可以使用 MapReduce 进行计算的简单而有趣的例子。

分布式 Grep：map 函数输出该行如果它与给定的模式匹配。Reduce 函数只需要将给定的中间结果输出。

url 访问频率计数：map 函数处理网页访问日志，输出<URL,1>。Reduce 函数将相同 URL 的 value 加起来，然后输出<URL,total count>对。

网页链接逆向图：map 函数输入<target,source>对表示从网页 source 到 target URL 的一条链接。Reduce 函数将给定 Target URL 的所有 source URL 连接到一块，然后输出<target,list(source)>。

Host 短语向量：一个 term vector 是指出现在一个文档或者文档集合中的最重要的单词的<word,frequency>对列表。Map 函数对每一个文档输出<hostname,term vector>(host 是从该文档对应的 url 中抽取出来)。Reduce 函数接受到一个给定 host 的所有文档的 term vector。它将这些 term vector 合并，并扔掉不常出现的那些

terms，然后输出一个最终的<hostname,term vector>对。

倒排索引：map 函数解析每个文档，输出一个<word,docid>对序列。Reduce 函数接受一个给定 word 的所有序列，对相应的 docid 进行排序，输出一个<word,list(docid)>。所有的输出对集合就形成了一个简单的倒排索引。通过很简单的改动，我们就可以让这个计算同时记住单词在文档中的出现位置。

分布式排序：map 函数从每条记录中提取 key，然后简单的输出<key,record>对。Reduce 函数原样地输出所有的对。该计算依赖于 4.1 节描述的划分功能以及 4.2 节描述的排序属性。

3.实现

对于 MapReduce 接口可以有很多不同的实现。正确的选择依赖于具体的环境。比如一个实现可能适用于小型共享内存机器，另一个可能适用于一个大的 NUMA 多处理机，另一个可能适用于更大的通过网络互联的集群。

本节描述一个面向 google 内部广泛使用的计算环境(由普通 pc 通过以太网交换机连接而成的大规模集群)的实现。在我们的环境里：

1. 机器主要是运行 linux 的双核 x86 处理器，每台机器具有 2-4GB 内存
2. 使用通用的网络硬件：在机器级别上，通常不是 100Mbps 就是 1Gbps，平均下来，整体的等分带宽要低些。
3. 集群由成百上千台机器组成，因此失败变得很普通
4. 存储是由连接到单个机器上的廉价的 IDE 硬盘提供的。一个内部开发的分布式文件系统被用来管理存储在硬盘上的数据。文件系统通过备份来为不可靠的硬件提供可用性和可靠性。
5. 用户提交 job 到调度系统。每个 job 由一组 task 组成，job 被调度系统映射到集群中的一组可用机器集合上去执行。

3.1 执行概览

通过自动将输入数据划分为 M 个片段，使得 Map 调用可以跨越多个机器执行。这些输入片段可以被不同的机器并行处理。Reduce 调用的分布，是通过使用一个划分函数(比如 hash(key) mod R)将中间结果的 key 的值域空间划分为 R 个片段。片段的个数 R 以及划分函数都是由用户描述的。

图 1 展示了一个 MapReduce 任务在我们的实现中的整体流程。当用户程序调用 MapReduce 函数时，将会产生如下的动作序列(图中的标号与如下描述中的数字相对应)：

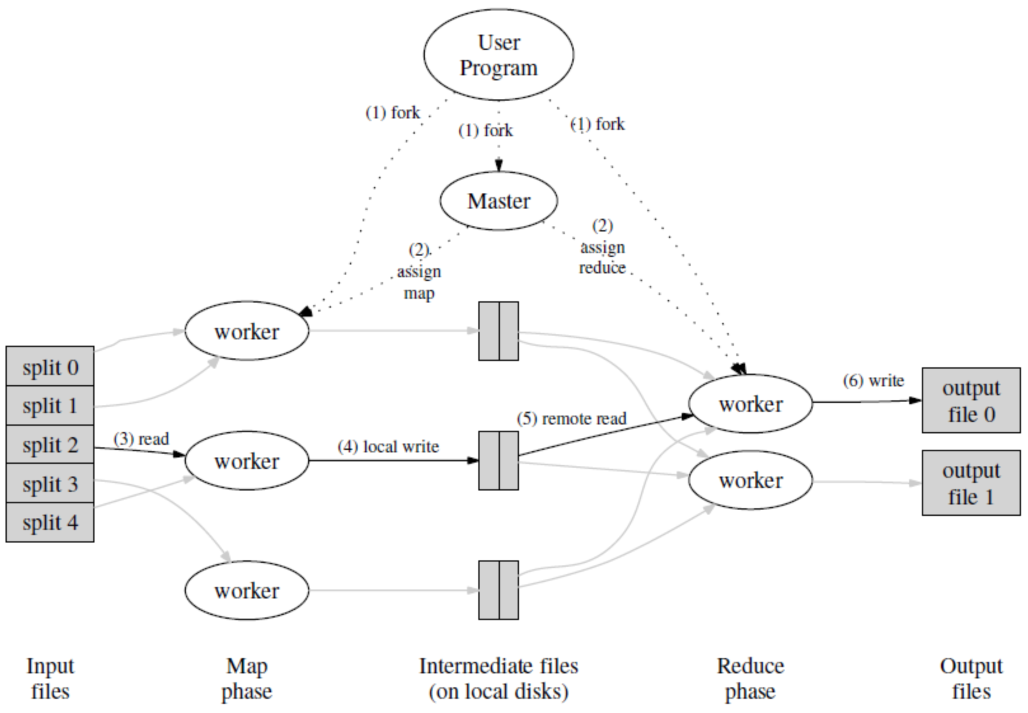


Figure 1: Execution overview

1. 用户程序中的 MapReduce 库首先将输入文件切分为 M 个片段(每个片段大小通常是 16MB 到 64MB，该大小用户可以通过一个配置参数控制)。然后在一组机器集上启动该程序的所有拷贝。
2. 在这些程序拷贝中有一个是特殊的：the master。其余的是称为 worker，由 master 为它们分配任务。总共有 M 个 map task 和 R 个 reduce task 需要分配。Master 选择空闲的 worker，给它们每个分配一个 map 或者 reduce task。
3. 被分配了 map task 的 worker 读取相应的输入片段内容。它从输入中解析出 key/value 对的集合，将每个对传递给用户定义的 map 函数处理。由 map 函数生成的中间结果被缓存在内存里。
4. 被缓存的那些对，通过划分函数被分成 R 个区域，然后周期性的被写入本地磁盘。然后将这些缓存对在本地磁盘上的位置返回给 master，master 再负责将这些位置信息传递给 reduce worker。
5. 当一个 reduce worker 被 master 通知了这些位置信息后，它就使用 RPC 调用去 map worker 的本地磁盘里读取这些缓冲数据。当一个 reduce worker 已经读取了所有的缓冲数据后，它就将它们根据 key 值进行排序，以让具有相同 key 值的被组织在一块。排序是需要的因为通常很多不同的 key 值会被映射到同一个 reduce task。如果中间结果的数据量太大以至于无法放入内存，还需要进行外排序。
6. Reduce worker 在排好序的中间结果数据上迭代，对于碰到的每个唯一的中间 key 值，它就将该 key 值，以及与它对应的 value 集合传递给用户定义的 reduce 函数。该 reduce 函数的输出会被 append 到这个 reduce worker 的最终输出文件上。
7. 当所有的 map task 和 reduce task 完成后，master 唤醒用户程序。这时用户程序从 MapReduce 调用里返回。

成功完成后，MapReduce 执行结果将会产生 R 个输出文件(一个 reduce task 对应一个，文件名由用户指定)。通常，用户不需要将这 R 个输出文件合并为一个文件，它们通常会作为另一个 MapReduce 调用的输入。或者通过另一个可以处理输入是划分为多个文件的分布式应用程序来使用它们。

3.2 master 数据结构

Master保存了几个数据结构。对于每个map和reduce task，它会保存它们的状态(*idle*, *in-progress*,or *completed* -- 空闲,处理中，完成)以及worker所在机器的标识(针对非空闲task)。{ !注意task与worker之间的关系，不是一对一的，一个worker可能处理多个task }

Master 是将中间结果文件位置从 map task 传递到 reduce task 的渠道。因此对于每个完成的(completed)map task，master 会保存由它产生的 R 个中间结果文件的大小及位置。当 map task 结束后，将会收到对于这些位置和大小信息的更新。这些信息又会被逐步推送给那些包含正在处理中的 reduce task 的 worker。

3.3 容错

因为 MapReduce 库是设计用来帮助在成百上千台集群上处理大量数据的，所以这个库就必须能够优雅地容忍机器失败。

Worker 失败

Master 周期性的 ping 每个 worker。如果在一定时间内没有收到某个 worker 的响应，就会把它标记为失败。由该 worker 执行完成的(completed)那些 map task{!注意 reduce task 不需要}都必须重置为空闲(idle)状态。这样，它们就又可以被调度到其他的 worker 上重新执行。类似的，那些该 woker 上执行中的(in-progress)任何 map reduce task{!注意 reduce task 也需要}，必须重置为空闲(idle)状态，重新加入调度。

已经完成的(completed)map task 需要重新执行，是因为它们的输出被存储在失败的那台机器的本地磁盘上，因此就变成了不可访问的。已经完成的(completed)reduce task 不需要重新执行，是因为它们的输出存放在一个全局文件系统上。

当一个 map task 首先由 work A 执行，然后又有 worker B 执行(因为 A 失败了)。所有正在执行(in-progress)reduce task 的 worker 都会收到该重新执行的通知。任何已经不能从 worker A 读数据的 reduce task 都将会从 worker B 读取。

MapReduce 可以很有效的应对大规模的 worker 失败。比如，在一个 MapReduce 任务期间，在一个运行中的集群上的网络维护可能导致 80 台机器几分钟内同时

无法访问。MapReduce master 简单的重新执行那些不可达机器上的任务。继续推进整个计算过程，最后完成该 MapReduce 任务。

Master 失败

很容易让 master 写入上面描述的数据结构的周期性检查点。如果 master 死掉后，就可以从上次的检查点状态开始启动一个新的拷贝。然而，由于只有一个 master，而且它失败的概率也比较低，因此在我们当前的实现中，如果 master 失败我们就结束 MapReduce 计算。在这种情况下，客户端可以进行检查，如果需要可以重试它们的 MapReduce 任务。

失败出现时的语义

当用户提供的 map 和 reduce 函数，针对它们的输入是一个确定性函数时，我们的分布式实现应该与整个程序串行执行时产生相同的输出。

我们通过 map 和 reduce task 输出的原子性提交来实现这个属性。每个执行中的 task 将它们的输出写入私有的 temp 文件里。一个 reduce task 产生一个这样的文件，一个 map task 产生 R 个这样的文件。当一个 map task 完成后，worker 会给 master 发送一个包含这个 R 个 temp 文件名称的消息。如果 master 收到一个已经完成的 map task 的完成消息，它会忽略该消息。否则，它会将这 R 个文件的名称记录在自己的一个数据结构中。

当 reduce task 完成后，reduce worker 会自动把 temp 输出文件重命名为最终的输出文件。如果相同的 reduce task 在多个机器上执行，多个重命名操作将会产生相同的最终输出文件。我们依赖于底层文件系统提供的原子性的重命名操作来保证最终的文件系统中只会包含一个 reduce task 执行产生的数据。

map 和 reduce 函数是确定性的以及等价于相同情况下串行执行结果的语义，主要优点是使得程序员可以很容易地理解程序的行为。当 map 和 reduce 操作不是确定性的时候，我们提供了虽然弱些但是仍然合理的语义。在出现不确定性操作时，一个特殊的 reduce task R1 的输出等价于该非确定性程序针对 R1 的串行执行输出。但是对于另一个 reduce task R2，就可能对应另一个不同的非确定性程序的串行执行结果。

考虑 map task M 和 reduce task R1 和 R2，令 e(Ri)代表已经完成的 Ri 的执行过程。语义可能会变得更弱，因为 e(R1)可能读取了 M 某次执行的输出结果，而 e(R2)可能读取了 M 另一次执行的输出。{首先 M 是不确定性的，其次 M 可能被重新执行过，这样 R1 R2 虽然读的是同一个 task 的输出，但是可能读取了不同的输出结果}。

3.4 本地化(locality)

在我们的计算环境中，网络带宽是相对宝贵的。我们通过利用输入数据(由 GFS

管理)是存储在机器的本地磁盘上的这一事实来节省网络带宽。GFS 将每个文件划分为 64MB 大小的块，每个块的几个副本存储在不同的机器上。MapReduce master 充分考虑输入文件的位置信息，尽量将一个 map task 调度到包含相应的输入数据副本的那个机器上。如果不行，就尝试将 map task 调度到该 task 的输入数据附近的那些机器(比如让 worker 所在的机器与包含该数据的机器在同一个网络交换机上)。当在一个集群上运行一个具有很多 worker 的大型 MapReduce 任务时，大部分的输入数据都是从本地读取的，很少消耗网络带宽。

3.5 任务粒度

如上所述，我们将 map 阶段划分为 M 个片段，将 reduce 阶段划分为 R 个片段。理想情况下，M 和 R 都应当远远大于运行 worker 的机器数目。让每个 worker 执行很多不同的 task 可以提高动态负载平衡，也能加速 worker 失败后的恢复过程：它已经完成的很多 map task 可以传给所有其他机器。在我们的实现中 M 和 R 到底可以多大，有一些实际的限制。因为 master 必须进行 O(M+R)个调度决定以及在内存中保存 O(M*R)个状态{!即每个 map task 的 R 个输出文件的位置信息，总共 M 个 task，所以是 M*R}。 (但是关于内存使用的常数因子是很小的：O(M*R)个状态，平摊下来大概每个 map task/reduce 对会占据一字节数据)。

另外，R 通常会被用户限制，因为每个 reduce task 的输出在不同的输出文件中。在实际中，我们通常这样选择 M：使每个独立 task 输入数据限制在 16MB 到 64MB 之间(这样上面所说的本地化优化是最有效的)。我们让 R 大概是我们将要使用的 worker 机器的几倍。我们通常这样执行 MapReduce 任务，在有 2000 个 worker 机器时，让 M = 20000,R = 5000。

3.6 备份任务 (Backup Tasks)

一个影响 MapReduce 任务整体执行时间的最常见的因素是”长尾”(花费相当长时间去完成 MapReduce 任务中最后剩下的极少数的那几个 task 的那台机器)。有很多原因可以导致长尾的出现。比如：具有一块坏硬盘的机器可能会经历频繁的可修正错误而使得 IO 性能从 30MB/s 降低到 1MB/s。集群调度系统可能会将那些引发 CPU、内存、本地磁盘或者网络带宽等资源竞争的 task 调度到同一台机器上。我们最近见过的一个错误是由于机器初始化代码中的一个 bug 引起的处理器缓冲失灵，使得受影响的机器上的计算性能降低了一百倍。

我们有一个可以缓解这种长尾问题的通用机制。当 MapReduce 任务接近尾声的时候，master 会备份那些还在执行中的 task。只要该 task 的主本或者其中的一个副本完成了，我们就认为它完成了。通过采用这种机制，我们只使计算资源使用率增长了仅仅几个百分点，但是明显地降低了完成整个 MapReduce 任务所需的时间。比如，在 5.3 节描述的排序例子中，如果不启用这个机制，整个完成时间将会增长 53%。

4.技巧

尽管通过简单书写 map 和 reduce 函数提供的基本功能对于我们大部分的应用来说足够了，我们也发现了其中的一些扩展也很有用。这一节，我们就来描述下它们。

4.1 划分函数

MapReduce 用户指定他们期望的 reduce task(也可以说输出文件)的数目 R。任务产生的数据通过在中间结果的 key 上使用一个划分函数被划分开。系统提供一个使用 hash 的默认的划分函数(比如 “hash(key) mod R”)。然而在某些情况下，使用关于 key 的其他函数进行划分更有用。比如有时候输入是 URL，我们希望来自相同 host 的输入可以存放在相同的输出文件上。为了支持这种情况，MapReduce 库的用户必须提供一个特殊的划分函数。比如使用”hash(Hostname(urlkey)) mod R” 作为划分函数，就可以让来自相同 host 的所有 URL 落在同一个输出文件上。

4.2 有序化保证

我们保证在一个给定的 partition 内，作为中间结果的 key/value 对是按照 key 值的增序进行处理的。这种有序化保证可以让每个划分的输出文件也是有序的。而这在输出文件格式需要支持按照 key 的有效的随机查找时非常有用，或者输出用户也会发现对这些数据进行排序会很方便。

4.3 合并函数

某些情况下，map task 产生的中间结果有很多具有相同 key 的重复值，而且用户指定的 reduce 函数又满足交换率和结合率。一个很好的例子就是 2.1 节里描述的 wordcount 的例子。因为单词频率的分布倾向于遵循 Zipf 分布，每个 map task 将会产生成百上千个相同的记录比如<the,1>这样的。而所有的这些又将会通过网络传递给一个 reduce task，然后通过 reduce 函数将它们累加起来。我们允许用户描述一个 combiner 函数，在数据通过网络发送之前对它们进行部分的归并。

Combiner 函数在每个执行 map task 的机器上执行。通常用来实现 combiner 和 reduce 函数的代码是相同的。唯一的不同在 MapReduce 库如何处理它们的输出。一个 reduce 函数的输出将会被写到最终的输出文件，而 combiner 函数的输出会被写到一个将要发送给 reduce task 的中间结果文件中。

4.4 输入和输出类型

MapReduce 库提供了几种不同格式的输入数据支持。比如”text”输入模式：将每

一行看做一个 key/value 对，key 是该行的 offset，value 是该行的内容。另一个支持的模式是一个根据 key 排序的 key/value 对的序列。每个输入类型知道如何将它们自己通过有意义的边界划分，然后交给独立的 map task 处理(比如 text 模式，会保证划分只会发生在行边界上)。用户可以通过提供一个 reader 接口的实现来支持新的输入类型。对于大多数用户来说，仅仅使用那些预定义的输入类型就够用了。

一个 reader 并不是必须从文件读数据。比如可以简单的定义一个从数据库或者是内存中的数据结构中读记录的 reader。

与之类似，我们也提供一组输出类型用于控制输出数据格式，同时用户也很容易添加对于新的输出类型的支持。

4.5 副作用(Side-effects)

MapReduce 的用户发现某些情况下，在 map 和 reduce 操作中顺便产生一个文件作为额外的输出会很方便。我们依赖于应用程序编写者来保证这些副作用的原子性以及幂等性{!因为 backup 机制可能导致一个 task 产生多个执行实例，而它们会写出同命的文件}。通常应用程序编写者会写一个 temp 文件，一旦它生成完毕再将它原子性的重命名。

我们并不为单个 task 产生的多个输出文件提供原子性的两阶段提交。因此那些具有跨文件一致性需求的产生多个输出文件的 task 应当是确定性的。这个限制在实际中还没有引起什么问题。

4.6 跳过坏记录

有时候用户代码中的一些 bug 会导致 Map 或者 Reduce 函数在处理某个特定记录时一定会 crash。这样的 bug 会使得 MapReduce 任务无法成功完成。通常的处理方法是修复这个 bug，但是有时候这样做显得并不灵活。因为 bug 可能是存在于第三方的库里，但是源代码是不可用的。而且有时候忽略一些记录是可以接受的，比如在一个大数据集上进行统计分析时。我们提供了一种可选的执行模式，在该模式下，MapReduce 库会检测哪些些记录引发了该 crash，然后跳过它们继续执行。

每个 worker 进程安装了一个信号处理器捕获那些段错误和总线错误。在调用用户 Map 或者 Reduce 操作之前，MapReduce 库使用一个全局变量存储该参数的序列号。如果用户代码产生了一个信号，信号处理器就会发送一个包含该序列号的"last gasp"的 UDP 包给 master。当 master 发现在同一记录上发生了不止一次失败后，当它在相应的 Map 或者 Reduce task 重新执行时，它就会指出该记录应该被跳过。

4.7 本地执行

在 Map 和 Reduce 函数上进行调试会变得很有技巧，因为实际的计算发生在分布式系统上，通常是几百台机器，而且工作分配是由 master 动态决定的。为了降低 debug，profile 的难度以及进行小规模测试，我们开发了一个 MapReduce 库的变更实现，让 MapReduce 任务的所有工作在本地计算机上可以串行执行。用户可以控制将计算放在特殊的 map task 上执行。用户通过使用一个特殊的 flag 调用它们的程序，然后就可以简单地使用他们的调试和测试工具(比如 gdb)。

4.8 状态信息

Master 运行一个内部的 http 服务器，然后发布一些用户可以查看的状态页面。这些状态页面展示了计算的进度，比如已经有多少任务完成，多少还在执行中，输入字节数，中间数据的字节数，输出的字节数，处理速率等等。该页面也会包含指向每个 task 的标准错误和标准输出文件的链接。用户可以使用这些数据来预测该计算还要花费多少时间，是否还需要为该计算添加更多的资源。计算远远低于预取时，这些页面也可以用来发现这些情况。

另外，更高级别的状态页会显示哪些 worker 失败了，当它们失败时正在处理哪些 map 和 reduce task。在诊断用户代码中的 bug 时，这些信息都是很有用的。

4.9 计数器

MapReduce 库提供了一些计数器设施来计算各种事件的发生。比如用户代码可能想计算处理的单词的总数，或者被索引的德语文档的个数等等。

为了使用这些设施，用户代码需要创建一个命名计数器对象然后在 Map 和/或 Reduce 函数中累加这些计数器。比如：

```
Counter* uppercase;
uppercase = GetCounter("uppercase");
map(String name, String contents):
for each word w in contents:
if (IsCapitalized(w)):
uppercase->Increment();
EmitIntermediate(w, "1");
```

来自各个 worker 机器的计数器的值将会周期性的发送给 master(通过对 master 的 ping 的响应捎带过去)。Master 将那些成功的 map 和 reduce task 的计数器值聚集，当 MapReduce 任务结束后，将它们返回给用户代码。当前的计数器值也会在 master 的状态页面上显示出来，这样用户就可以看到计算的实时进展。在计算计数器值时，master 会忽略掉那些重复执行的相同 map 或者 reduce task 的值，以避免重复计数。(重复执行可能是由于备份任务的使用或者是 task 失败引

发的重新执行而引起的)

一些计数器值是由 MapReduce 库自动维护的，比如已经处理的输入 key/vaule 对的个数，已经产生的输出 key/vaule 对的个数。

用户发现计数器设施对于 MapReduce 任务的行为的完整性检查是非常有用的。比如，在某些 MapReduce 任务中，用户代码可能想确定已产生的输出对的数目是否刚好等于已处理的输入对数目，或者已经被处理的德语文档在已处理的文档中是否在一个合理的比例上。

5.性能

在本节中我们将通过运行在大规模集群上的两个计算任务来测量 MapReduce 的性能。一个计算是在大概 1TB 的数据中搜索给定模式的文本。另一个计算是对接近 1T 的数据进行排序。

这两个程序就可以代表 MapReduce 用户所写的实际程序中的大部分子集：一类是将数据从一种表现形式转换为另一种表现形式的程序，另一类就是从一个大数据集合中抽取少量感兴趣的数据集。

5.1 集群配置

所有的程序都是在一个由将近 1800 台机器组成的集群上执行。每台机器有 2 个打开了超线程的 2G Intel Xeon 处理器，4GB 内存，2 个 160GB IDE 硬盘，一个 gigabit 以太网链路。这些机器通过一个两级的树形交换网络连接，根节点具有接近 100-200 Gbps 的总体带宽。所有机器具有相同的配置，在任意两个机器间的往返时间小于 1ms。

在 4GB 内存中，大概 1-1.5G 内存预留给在集群上运行的其他 task。程序在某个周末的下午执行，此时 cpu、硬盘、网络接近空闲。

5.2Grep

Grep 程序通过扫描 10^10 个 100 字节的记录，查找一个很少出现的三字符模式(该模式出现在 92337 个记录里)。输入被划分为近似 64MB 大小的片段(M=15000)，整个输出被放在一个文件中(R=1)。

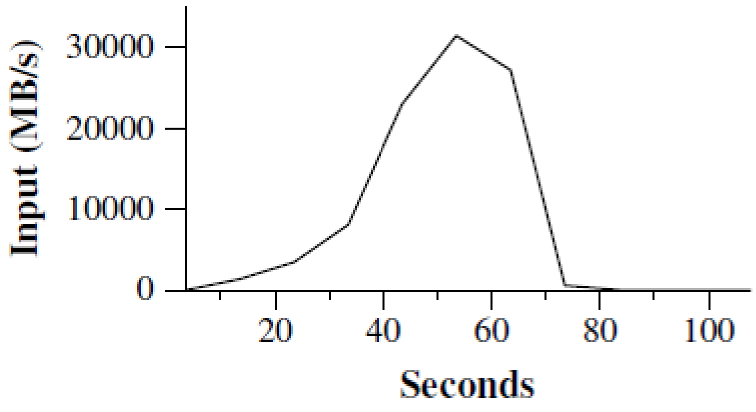


Figure 2: Data transfer rate over time

图 2 展示了整个计算的处理过程。Y 轴表示输入数据的扫描速率。伴随着用于进行该 MapReduce 任务的机器数的增多，该速率也在逐渐攀升，当有 1764 个 worker 被分配该任务后达到了 30GB/s 的峰值。当 map task 结束后，该速率开始下降，大概在 80 秒的时候基本上降为 0。整个计算过程花费了接近 150 秒，这包括一分钟的启动时间(这个开销主要是由将程序传输给所有 worker，与 GFS 交互以打开 1000 个输入文件以及得到本地化优化所需要的信息造成的)。

5.3 排序

排序程序对 10^10 个 100 字节的记录进行排序(接近 1TB 数据)。这个程序的模型源自于 TeraSort Benchmark。

排序程序总共由不到 50 行用户代码组成。Map 函数只有 3 行，将 10 字节长的排序用 key 值从一个文本行中抽取出来，然后输出该 key，以及原始的文本行，作为中间结果 key/value 对。我们使用内建的 Identity 函数作为 reduce 操作。该函数将中间结果不过任何改变地输出。最后的排好序的结果写到一个具有 2 个副本的 GFS 文件集合上(即该程序将会产生 2TB 的输出)。

与之前的类似，输入数据被划分为 64MB 的片段(M=15000)。排好序的输出被划分为 4000 个输出(R=4000)。划分函数使用 key 的字节表示来将它们划分为 R 个片段。

对于该 benchmark 的划分函数建立在对于 key 值分布的了解上。对于一个通常的排序问题里，我们会增加一个预先进行的 MapReduce 操作，该操作会收集 key 值的采样值，然后使用这些 key 值的采样来计算最终排序序列的划分点。

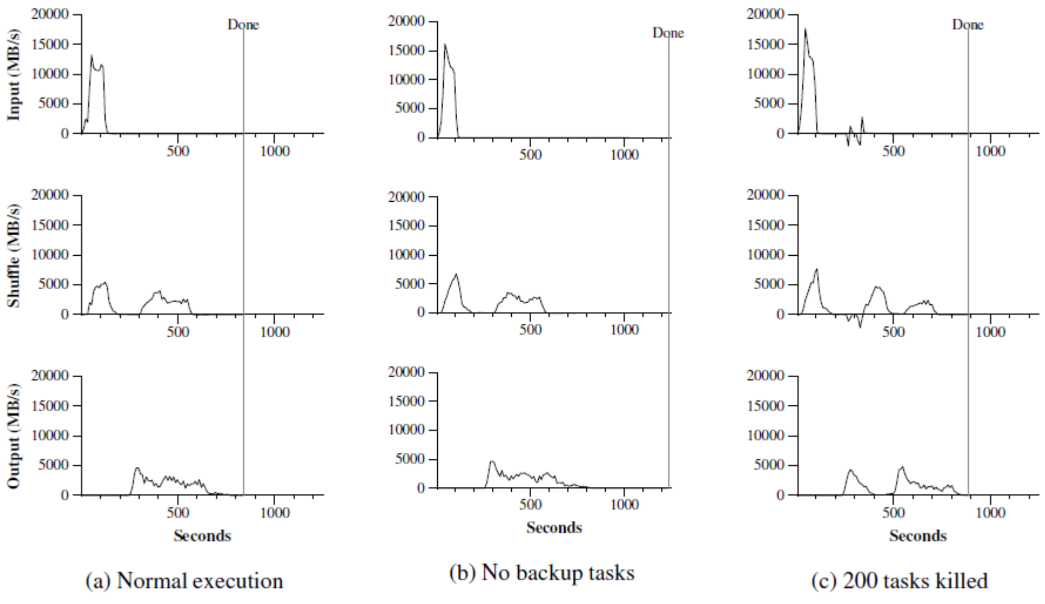


Figure 3: Data transfer rates over time for different executions of the sort program

图 3(a)展示了该排序程序的一个正常的处理过程。左上角的图表示输入速率。峰值速率大概是 13GB/s，由于所有的 map task 在 200 秒前都结束了，所以该速率下降的很快。可以看到，输入速率要小于 grep。这是因为排序的 map task 花费了大概一半的时间和 IO 带宽将中间结果写入到本地硬盘上。而与之相比，grep 的中间结果输出几乎可以忽略不及。

左边中间的图展示了数据从 Map task 向 reduce task 的网络传输速率。当第一个 map task 完成后，shuffling 就开始了。图中的第一个峰值是由于第一批的 1700 个 reduce task 都启动后产生的(整个 MapReduce 任务被分配给大概大概 1700 个机器，每个机器同一时刻最多执行一个 Reduce task)。大体上在 300 秒的时候，这一批的 reduce task 结束，然后启动了剩余的 reduce task 的 shuffling 过程。大概在 600 秒时，这些 shuffling 才结束。

左边最底下的图形展示了 reduce task 将排好序的数据写入最终输出文件的速率。在第一次的 shuffling 的结束与数据写入开始之间存在一个延时，是因为机器此时正忙着对中间数据进行排序。写入过程在一段时间内大概持续着大概 2-4GB/s 的速率。所有的写入大概在 850 秒的时候结束。假设启动的花费，整个计算花费了 891 秒。这接近于当前 Terasort benchmark 的最快结果 1057 秒。

另外需要指出的是：输入速率比 shuffle 速率和输出速率高是因为我们的本地化优化(大部分的数据都是从本地硬盘读取的，这就绕过了网络带宽的限制)。Shuffle 速率比输出速率高是因为输出阶段要写两份拷贝(保存两份是为了可靠性和可用性){这两份拷贝是需要耗费网络带宽的}。写两个副本是因为这是我们的底层文件系统提供的可靠性和可用性机制。如果底层文件系统使用了 erasure 编码{!一种可纠错编码方式，在数据失效时，可以结合校验编码恢复出原始数据}而不是副本，对于写数据的网络带宽需求将会减少。

5.4 任务备份的影响

在图 3(b),我们展示了一个没有开启任务备份的排序程序的执行过程。执行流类似于我们在图 3(a)里看到的那样。除了在繁重的写活动出现后出现了一个长尾。在 960 秒时，只剩下 5 个 reduce task 还没有完成。然而这些掉队者，在 300 秒后才完成。整个计算花费了 1283 秒，增加了 44%。

5.5 机器失败

图 3(3),展示了我们在计算执行几分钟后，杀掉 1746 个 worker 里面的 200 个后的执行过程。底层的集群调度器，立刻重启在这些机器上的 worker 进程(因为只是进程被杀掉了，机器仍然是可用的)。

死掉的 worker 作为负的输入速率进行显示，因为前面以及完成的 map task 的工作都消失了需要重新执行。Map task 的重新执行相对较快。加上启动时间，整个计算过程在 933 秒的时候结束，仅仅比正常情况下的执行时间增加了 5%。

6.经验

在 2003 年 2 月，我们写出了第一版的 MapReduce 库，2003 年 8 月对它进行了很多包括本地化(locality)优化，跨机器的 task 执行的动态负载平衡等等在内的改进。从那时起，我们欣喜的发现 MapReduce 库可以如此广泛地应用在我们工作中的各种问题上。目前它已经在 google 内部应用在广泛的领域上：

- 大规模机器学习问题
- 用于 Google 新闻和购物的聚类问题
- 找到最流行的查询词
- 为了实验或者产品从网页中抽取属性(比如为了本地化搜索从大量网页中抽取地理位置)
- 大规模图计算

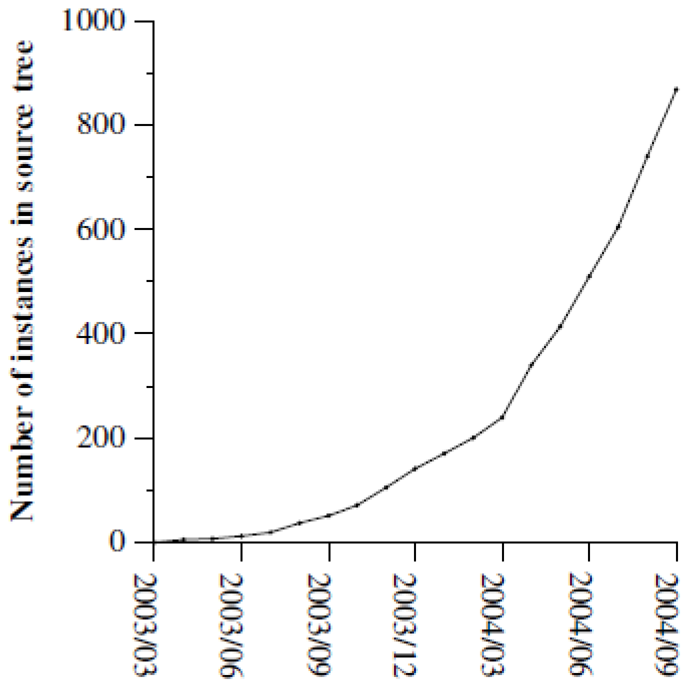
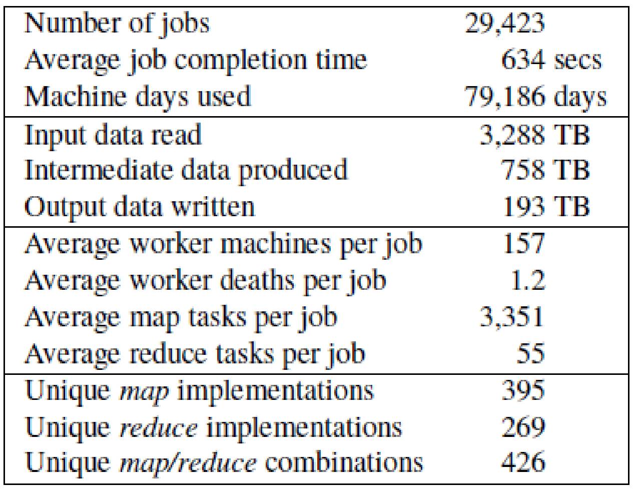


Figure 4: MapReduce instances over time

图 4 展示了过去的时间里，提交到我们的源代码管理系统中的 MapReduce 程序的数目。从 2003 年早期的 0 个到 2004 年 9 月接近 900 个。MapReduce 之所以如此成功，是因为它使得在半小时内写出一个简单地可以在数千台机器上运行的程序成为可能。这大大加速了我们的开发和原型周期。此外，它还使得没有分布式或者并行系统编程经验的程序员可以很容易地使用大量的计算资源。

在每个 job 结束时，MapReduce 库还会记录该 job 使用的计算资源的统计信息。表 1，我们展示了 2004 年 8 月，在 google 内部运行 MapReduce job 的一个子集的一些统计信息。



Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB
Average worker machines per job	157
Average worker deaths per job	1.2
Average map tasks per job	3,351
Average reduce tasks per job	55
Unique *map* implementations	395
Unique *reduce* implementations	269
Unique *map/reduce* combinations	426

Table 1: MapReduce jobs run in August 2004

6.1 大规模索引(Large-Scale Indexing)

目前为止，我们一个最重要的 MapReduce 应用就是用它完全重写了产品索引系统，该系统为 google 的网页搜索服务产生所需要的数据结果。索引系统以一个由爬虫抓取的存储在 GFS 上的很大的文档集合作为输入，总共数据量要超过 20TB。索引流程由 5 到 10 个 MapReduce 任务组成。通过使用 MapReduce(而不是使用之前版本的索引系统所使用的自适应的分布式传输)有如下几个优点：

- 索引代码很简单，少而且容易理解。因为用于容错、分布和并行化的代码都隐藏在了 MapReduce 库中。比如，我们通过使用 MapReduce 将原来的一个计算过程的代码量从 3800 行降低到了 700 行。
- MapReduce 库的性能已经足够好了，这样我们就能将不相关地计算分离，而不是为了降低额外的传输费用而将它们合在一块。这使得我们很容易改变索引处理过程。比如过去在旧系统中可能需要几个月才能完成的变更，现在在新的系统中几天就可以完成。
- 索引处理流程变得很容易操作。因为大部分由于机器失败、慢机器以及网络引发的问题都由 MapReduce 库自动处理掉了，不需要进行额外的干预。另外也很容易通过给索引系统增加新机器来提高性能。

7.相关工作

已经有很多系统提供了严格的编程模型，使用了很多限制来进行计算的并行化。MapReduce 模型可以看做是基于我们在现实中的海量计算经验，对这些模型的一个简化和提炼。更重要的是，我们提供了一个可以扩展到数千个处理器上的容错实现。与之相比，大部分的并行处理系统只是在小规模集群上实现的，将机器错误处理交给程序员。

大同步模型和一些 MPI 实现为简化程序员编写并行程序提供了更高级别的抽象。这些系统与 MapReduce 的一个关键不同就是 MapReduce 使用了一个限制性的编程模型来为用户程序提供自动地并行化和透明的容错机制。

我们的本地化优化策略主要源于这样的一些技术，比如 active disks[12,15]，在那里为了降低 IO 或者网络的数据传输，计算被放到那些靠近本地硬盘的处理元素中执行。我们是在由少量硬盘直接连接的 PC 上运行而不是在一个磁盘控制处理器上运行，但是策略是类似的。

我们的任务备份机制类似于 Charlotte 系统[3]中使用的 eager 调度机制。简单 eager 调度机制的一个缺点是如果给定的 task 引发了重复的失败，整个计算就无法完成。我们通过跳过坏记录的方式解决了这样的问题。

MapReduce 实现依赖于内部开发的一个集群管理系统，它负责在一个机器集合上分布调度用户任务。尽管不是本文关注的重点，该集群管理系统类似于 Condor[16]。

作为 MapReduce 库的一部分的排序设施在操作过程上类似于 Now-sort[1]。源机器(map task)将数据划分进行排序，然后将每份传递给一个 R 个 reduce worker 中的一个。每个 reduce worker 在本地进行排序(如果可能的话就仅使用内存排序)。当然 NOW-sort 并不包含使得我们的库应用广泛的 Map 和 Reduce 函数。

Rive[2]提供了一个进程间通过分布式队列进行数据传输的编程模型。像 MapReduce 一样，River 尽量提高系统的平均性能，即使是由于硬件异构或者系统扰动出现了非对称的情况。River 通过细致的硬盘和网络传输调度来达到均衡的完成时间。MapReduce 使用了不同的策略。通过限制编程模型，MapReduce 框架能将问题划分为大量的细粒度 task。这些 task 可以在可用的 worker 上进行动态的调度，这样跑的快的 worker 就可以处理更多的 task。该编程模型也允许我们在 job 快结束的时候调度 task 进行冗余的执行，这样大大减少了长尾出现时的完成时间。

BAD-FS[5]有一个与 MapReduce 完全不同的编程模型。与 MapReduce 不同，它的目标是降低在广域网上的 job 的执行时间。但是，它们具有两个基本的相同点：1.都采用了冗余执行的方式，从导致数据丢失的失败中快速恢复 2.都采用了本地化优化以降低数据在网络上的传输。

TACC[7]是一个设计用于简化构建高可用网络服务的系统。与 MapReduce 类似，它依赖于重执行作为实现容错的一个机制。

8.总结

MapReduce 编程模型已经因各种目的在 google 内部成功使用。我们将这种成功

归为几个原因。首先，模型很容易使用，即使对于没有分布式编程经验的程序员来说也是，因为它隐藏了并行化、容错、本地化优化、负载平衡的细节。第二，大量的问题可以简单地用 MapReduce 计算来表达。比如 MapReduce 被用来为 google 的网页搜索服务、排序、数据挖掘、机器学习等很多其他的系统生成数据。第三，我们开发了一个可以扩展到数千台机器上 MapReduce 实现。该实现可以充分利用机器的资源，因此很适合用来处理在 google 碰到的很多大规模计算问题。

通过这项工作我们也学到了很多。首先，通过限制编程模型可以使计算的并行化和分布很简单，同时也能让它容错。第二，网络带宽是一种稀缺资源。我们系统中大量的优化都是为了降低网络传输数据量：本地化优化允许我们从本地磁盘上读数据，将单份拷贝的中间数据写入本地磁盘节省了网络带宽。第三，冗余执行能用来降低慢机子的影响，以及用来处理机器失败和数据丢失。

致谢

……
MapReduce 从 GFS 上读取输入数据以及写出输出数据，因此我们要感谢…在开发 GFS 上的工作…我们还要感谢…在开发 MapReduce 使用的集群管理系统上的工作……

Bigtable:结构化数据的分布式存储系统

作者：Fay Chang、Jeffrey Dean 、Sanjay Ghemawat etc. Google Inc 2006
译者：phylips@bmy 2010-10-6
出处：http://duanple.blog.163.com/blog/static/709717672010961173782/

摘要

Bigtable 是设计用来管理那些可能达到很大大小(比如可能是存储在数千台服务器上的数 PB 数据)的结构化数据的分布式存储系统。Google 的很多项目都将数据存储在 Bigtable 中，比如网页索引、Google 地球、Google 金融。这些应用对 Bigtable 提出了很多不同的要求，无论是数据大小(从单纯的 URL 到包含图片附件的网页)还是延时需求。尽管存在这些各种不同的需求，Bigtable 成功地为 Google 的所有这些产品提供了一个灵活的，高性能的解决方案。在这篇论文中，我们将描述 Bigtable 所提供的允许客户端动态控制数据分布和格式的简单数据模型，此外还会描述 Bigtable 的设计和实现。

1. 导引

在过去的 2 年半时间里，我们设计、实现、部署了一个称为 Bigtable 的用来管理 Google 的数据的分布式存储系统。Bigtable 的设计使它可以可靠地扩展到 PB 级的数据以及数千台机器上。Bigtable 成功地实现了这几个目标：广泛的适用性、可扩展性、高性能以及高可用性。目前，Bigtable 已经被包括 Google 分析、Google 金融、Orkut、个性化搜索、Writely 和 Google 地球在内的 60 多个 Google 产品和项目所使用。这些产品使用 Bigtable 用于处理各种不同的工作负载类型，从面向吞吐率的批处理任务到时延敏感的面向终端用户的数据服务。这些产品所使用的 Bigtable 集群也跨越了广泛的配置规模，从几台机器到存储了几百 TB 数据的上千台服务器。

在很多方面，Bigtable 都类似于数据库：它与数据库采用了很多相同的实现策略。目前的并行数据库和主存数据库已经成功实现了可扩展性和高性能，但是 Bigtable 提供了与这些系统不同的接口。Bigtable 并不支持一个完整的关系数据模型，而是给用户提供了一个可以动态控制数据分布和格式的简单数据模型，允许用户将数据的 locality 属性体现在底层的数据存储上。数据使用行和列的名称(它们可以是任意字符串)进行索引。Bigtable 将数据看做是未经解释的字符串，尽管用户经常将各种形式的结构化或半结构化的数据存储到这些字符串里。用户可以通过细心设计他们的 schema，来控制数据的 locality。最后，Bigtable 的 schema 参数还允许用户选择从磁盘还是内存获取数据。

第 2 节更加详细的描述了该数据模型。第 3 节提供了关于用户 API 的概览。第 4 节简要描述了 Bigtable 所依赖的底层软件。第 5 节描述了 Bigtable 的基本实现。第 6 节描述了我们为提高 Bigtable 的性能使用的一些技巧。第 7 节提供了一些对

于 Bigtable 的性能测量数据。第 8 节展示了几个 Google 内部的 Bigtable 使用实例。第 9 节讨论了我们在设计和为 Bigtable 应用提供支持的过程中，所学到的一些经验教训。最后第 10 节描述了相关工作，第 11 节进行了总结。

2. 数据模型

Bigtable 是一个稀疏的，分布式的一致性多维有序 map。这个 map 是通过行关键字，列关键字以及时间戳进行索引的；map 中的每个值都是一个未经解释的字节数组。

(row:string,column:string,time:int64)　-> string

我们在对于这种类 Bigtable 系统的潜在使用场景进行了大量考察后，最终确定了这个数据模型。举一个影响到我们某些设计决策的具体例子，比如我们想保存一份可以被很多工程使用的一大集网页及其相关信息的拷贝。我们把这个表称为 webtable，在这个表中，我们可以使用 URL 作为行关键字，网页的各种信息作为列名称，将网页的内容作为表的内容存储：获取的时候还需要在列上加上时间戳，如图 1 所示。

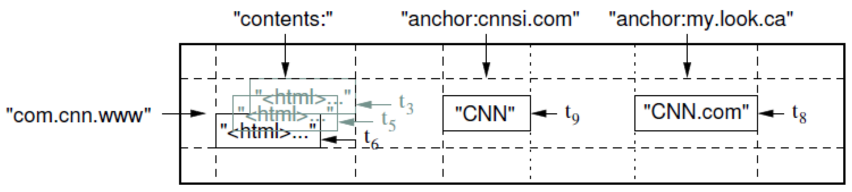


Figure 1: A slice of an example table that stores Web pages. The row name is a reversed URL. The contents column family contains the page contents, and the anchor column family contains the text of any anchors that reference the page. CNN's home page is referenced by both the Sports Illustrated and the MY-look home pages, so the row contains columns named anchor:cnnsi.com and anchor:my.look.ca. Each anchor cell has one version; the contents column has three versions, at timestamps t_3, t_5, and t_6.

表中的行关键字是大字符串(目前最大可以到 64KB，尽管对于大多数用户来说最常用的是 10-100 字节)。在一个行关键字下的数据读写是原子性的(无论这一行有多少个不同的列被读写)，这个设计使得用户在对相同行的并发更新出现时，更容易理解系统的行为。

Bigtable 按照行关键字的字典序来维护数据。行组{!row range，将它翻译为行组，一个 row range 可能由多个行组成}是可以动态划分的。每个行组叫做一个 tablet，是数据存放以及负载平衡的单位。这样，对于一个短的行组的读就会很有效，而且只需要与少数的机器进行通信。客户端可以通过选择行关键字来利用这个属性，这样它们可以为数据访问得到好的 locality。比如，在 webtable 里，相同域名的网页可以通过将 URL 中的域名反转而使他们放在连续的行里来组织到一块。比如我们将网页 maps.Google.com/index.html 的数据存放在关键字 com.Google.maps/index.html 下。将相同域名的网页存储在邻近位置可以使对主机或域名的分析更加有效。

列族(Column Families)

不同的列关键字可以被分组到一个集合，我们把这样的一个集合称为一个列族，

它是基本的访问控制单元。存储在同一个列族的数据通常是相同类型的(我们将同一列族的数据压缩在一块)。在数据能够存储到某个列族的列关键字下之前，必须首先要创建该列族。我们假设在一个表中不同列族的数目应该比较小(最多数百个)，而且在操作过程中这些列族应该很少变化。与之相比，一个表的列数目可以没有限制。

一个列关键字是使用如下的字符来命名的：family:qualifier。列族名称必须是可打印的,但是 qualifier 可能是任意字符串。比如 webtable 有一个列族是 language，它存储了网页所使用的语言。在 language 列族里，我们只使用了一个列关键字，里面存储了每个网页的 language id。该表的另一个列族是 anchor，在该列族的每个列关键字代表一个单独的 anchor，如图 1 所示。Qualifier 是站点的名称，里面的内容是链接文本。

访问控制以及磁盘和内存分配都是在列族级别进行的。在 webtable 这个例子中，这些控制允许我们管理不同类型的应用：一些可能会添加新的基础数据，一些可能读取这些基础数据来创建新的列族，一些可能只需要查看现有数据(甚至可能因为隐私策略而无法查看所有现有数据)。

时间戳

Bigtable 里的每个 cell 可以包含相同数据的多个版本；这些不同的版本是通过时间戳索引的。Bigtable 的时间戳是一个 64 位的整数。它们可以由 Bigtable 来赋值，在这种情况下它们以毫秒来代表时间。也可以由客户端应用程序显式分配。应用程序为了避免冲突必须能够自己生成唯一的时间戳。一个 cell 的不同版本是按照时间戳降序排列，这样最近的版本可以被首先读到。

为了使不同版本的数据管理更简单，我们支持通过 2 种方法针对每个列族进行设定，来告诉 Bigtable 如何对 cell 中的数据版本进行自动的垃圾回收：用户可以指定最近的哪几个版本需要保存，或者保存那些足够新的版本(比如只保存那些最近 7 天写的数据)。

在我们的 webtable 中，我们将被抓取网页的时间戳存储在 contents 里：这些时间说明了这些网页的不同版本分别是在何时被抓取的。前面描述的垃圾回收机制，可以使我们只保存每个网页最近的三个版本。

3. API

Bigtable API 提供了一些函数用于表及列族的创建和删除。它也提供了一些函数用于改变集群，表格及列族的元数据，比如访问控制权限。

客户端应用程序可以写入或者删除Bigtable里的值，从行里查找值或者在表中的一个数据子集中进行迭代。图 2 展示了使用RowMutation执行一系列更新的C++代码(为了保持简单省略了一些不相关的细节)。Apply调用对webtable执行了一个

原子性的变更操作：给 www.cnn.com增加一个anchor，然后删除另一个anchor。

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

Figure 2: Writing to Bigtable.

图 3 展示的 c++代码使用 Scanner 来在一个特殊行上的所有 anchor 进行迭代，用户可以在多个列族上进行迭代，存在几种机制来对扫描到的行、列、时间戳进行过滤。比如我们可以限制只扫描那些与正则表达式"anchor:*.cnn.com"匹配的列，或者那些时间戳距离当前时间 10 天以内的 anchor。

```
Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily("anchor");
stream->SetReturnAllVersions();
scanner.Lookup("com.cnn.www");
for (; !stream->Done(); stream->Next()) {
  printf("%s %s %lld %s\n",
         scanner.RowName(),
         stream->ColumnName(),
         stream->MicroTimestamp(),
         stream->Value());
}
```

Figure 3: Reading from Bigtable.

Bigtable 提供了其他的几种 feature 允许用户使用更复杂的方式熟练控制数据。首先，Bigtable 支持单行事务，能够支持对存储在一个行关键字上的数据执行原子性的读-修改-写序列。Bigtable 目前并不支持普通的跨行事务，尽管它提供了一个多个用户的跨行写的接口。其次，Bigtable 允许用户将 cell 作为一个整数计数器来使用。最后，Bigtable 支持在服务器地址空间内执行一个客户端脚本。这些脚本是使用 Google 内部开发的数据处理语言 Sawzall 编写的。当前，我们基于 Sawzall 的 API 不允许客户端脚本向 Bigtable 中写回，但是它允许进行各种形式的数据转换，基于各种表达式的过滤以及大量的统计操作符。

Bigtable 可以与 MapReduce(Google 内部开发的一个运行大规模并行程序的框架)一起使用。我们写了很多 wrapper，以使得 Bigtable 可以作为 MapReduce job 的输入源或者输出目标。

4. 基础构件

Bigtable 是建立在 Google 的其他几个设施之上。Bigtable 使用 GFS 来存储日志和数据文件。Bigtable 集群通常运行在一个运行着大量其他分布式应用的共享机器池上。Bigtable 依赖于一个集群管理系统进行 job 调度和共享机器上的资源管理，

处理机器失败以及监控机器状态。

Bigtable内部采用Google SSTable文件格式来存储数据。一个SSTable提供了一个一致性的，有序的从key到value的不可变map，key和value都是任意的字节串。操作通常是通过一个给定的key来查找相应的value，或者在一个给定的key range上迭代所有的key/value对。每个SSTable内部包含一系列的块(通常每个块是64KB大小，该大小是可配置的)。一个块索引是用来定位block的(保存在SSTable的尾部){!为何存放在尾部？实际上我们可以想象一下该文件的存储方式，用户在刚开始写入的时候，如果不对需要写入的数据进行一次遍历，它是不知道需要写入的数据大小的，也无法知道它们在文件中的offset位置，也就无法生成索引，所以索引信息是当用户写入数据完毕之后，才会得到完整的信息，因此索引信息是放在尾部的，用户写完数据后，就同时得到了索引信息，这时就可以继续写索引信息了。另外参考HFile: A Block-Indexed File Format to Store Sorted Key-Value Pairs的关于HFile的Close操作需要完成的工作，也可以看出应该如何安排SSTable内部的数据顺序}，当SSTable打开时该索引会被加载到内存。一次查找可以通过一次磁盘访问完成：首先通过在内存中的索引进行一次二分查找找到相应的块，然后从磁盘中读取该块。另外，一个SSTable可以被完全映射到内存，这样就不需要我们接触磁盘就可以执行所有的查找和扫描。{!关于SSTable(StaticSearchTable)的具体格式可以参考HFile存储格式中对HBASE的HFile的介绍}

Bigtable 依赖于一个高可用的一致性分布式锁服务 Chubby。Chubby 由 5 个活动副本组成，其中的一个选为 master 处理请求。当超过半数的副本运行并且可以相互通信时，该服务就是可用的。Chubby 使用 Paxos 算法来保证在出现失败时，副本的一致性。Chubby 提供了一个由目录和小文件组成的名字空间。每个文件或者目录可以当作一个锁来使用，对于一个文件的读写是原子性的。Chubby 的客户端库为 Chubby 文件提供一致性缓存。每个 Chubby 客户端维护着一个与 Chubby 服务的会话。如果在租约有效时间内无法更新会话的租约，客户端的会话就会过期。当一个客户端会话过期后，它会丢失所有的锁和打开的文件句柄。Chubby 客户端也会在 Chubby 文件和目录上注册回调函数来处理这些变更或者会话的过期。

Bigtable 使用 Chubby 来完成各种任务：保证任意时刻最多只有一个活动的 master；存储 Bigtable 数据的 bootstrap location(参见 5.2 节)；发现 tablet 服务器以及确定 tablet 服务器的死亡(参见 5.2 节)；保存 Bigtable schema 信息(每个表的列族信息)；存储访问控制列表。如果 Chubby 在一段时间内不可用，Bigtable 也会不可用。我们最近在使用了 11 个 Chubby 实例的 14 个 Bigtable 集群进行了测量。由于 Chubby 不可用而造成的存储在 Bigtable 上的数据不可用的平均概率是 0.0047%。对于单个集群来说，由于 Chubby 不可用造成的这个概率是 0.0326%。

5. 实现

Bigtable 实现由 3 个主要的组件构成：每个客户端需要链接的库，一个 master 服务器，很多 tablet 服务器。可以从一个集群中动态添加(或者删除) tablet 服务器来适应工作负载的动态变化。

Master 负责将 tablet 分配到 tablet 服务器，检测 tablet 服务器的添加和过期，平衡 tablet 服务器负载，对存储在 GFS 上的文件进行垃圾回收。另外，它还会处理 schema 的变化，比如表和列族的创建。

每个 tablet 服务器管理一个 tablets 集合(通常每个 tablet 服务器有 10 到 1000 个 tablet)。Tablet 服务器负责它已经加载的那些 tablet 的读写请求，也会将那些过于大的 tablet 进行分割。{!tablet 服务器本身实际上是 GFS 的用户，它们只是负责它加载的那些 tablet 的管理，这些 tablet 的物理存储并不一定存放在管理它的 tablet 服务器上，底层的存储是由 GFS 完成的，tablet 服务器可以只调用它的接口来完成相应任务。而 METADATA 表中的位置信息应该是指某个 tablet 由哪个 tablet 服务器管理，而不是物理上存储在哪个机器上。这样看来 Bigtable 充当了客户端与 GFS 间的读写代理}

正如很多单 master 的分布式存储系统，客户端数据的移动并不会经过 master：客户端直接与 tablet 服务器进行通信来进行读写。因为 Bigtable 客户端并不依赖于 master 得到 tablet 的位置信息，大部分的客户端从来不会与 master 通信。所以，master 实际中通常都是负载很轻的。

Bigtable 集群存储了大量的表。每个表由一系列的 tablet 组成，每个 tablet 包含一个行组的所有相关数据。一开始，每个表由一个 tablet 组成。随着表格的增长，它会自动分割成多个 tablet，它们大小默认是 100-200MB。

5.1 Tablet 放置

我们使用一个类似于 B+树的三级结构来存储 tablet 的放置信息如图 4。

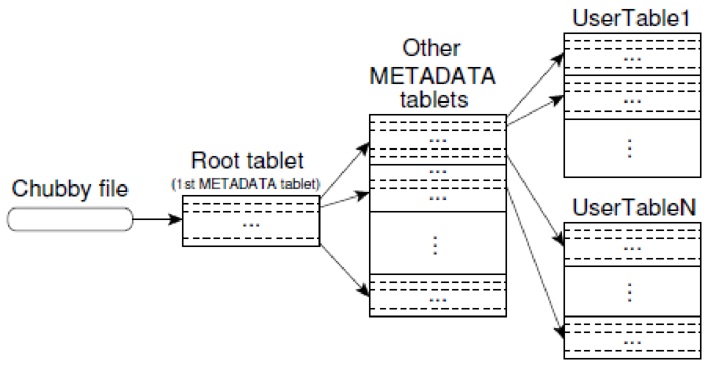


Figure 4: Tablet location hierarchy.

第一级是一个存储在 Chubby 中的包含了 root tablet 位置信息的文件。root tablet 包含了在一个特殊的 **METADATA** 表里的所有 tablet 的位置信息{!root tablet 实际上是 METADATA 表的第一个 tablet，它存储了该表其他的 tablet 的位置信息}。每个 METADATA tablet 包含了一组用户 tablet 的位置。Root tablet 仅仅是 **METADATA** 表的第一个 tablet，但是是特殊对待的-它永远不会被分割-为了保证 tablet 位置信息的层次结构不会超过 3 级。

METADATA 表的每一个行关键字(由 tablet 所属的表标识符和它的结束行组成)下存储了一个 tablet 的位置。每个 METADATA 行在内存中大概存储了 1KB 数据。我们限制 **METADATA 的 tablet** 的大小为 128MB，我们的三级层次结构足以用来寻址 2^34 个 tablet(如果 tablet 按照 128MB 算，就是 2^61 字节){!root tablet 大小为 128M，每个行 1KB，那么它就可以存储 128*2^20/2^10=128*2^10 个 METADATA tablet，同样的，每个 METADATA tablet 可以存储 128*2^10 个普通 tablet，这样总共可以存储 128*2^10*128*2^10 即 2^34 个普通 tablet，每个 tablet 有将近 1KB 元数据，这样算起来存储这些元信息就需要 4TB 的数据，所以该 METADATA 表是分布式的放在不同的机器上的，同时也不可能全部放入内存，而是采用与普通的表一样的存储方式，放在 GFS 上。但是会把某些特殊信息放在内存中，比如第 6 节提到的：METADATA 中的 location 列族会被放入内存 }。

客户端库会缓存 tablet 的位置信息。如果客户端不知道某个 tablet 的信息，或者发现缓存的位置信息是错误的，那么它就会递归地在 tablet 位置存储结构中查找。如果客户端缓存是空的，定位算法需要三次网络往返，包括从 Chubby 的一次读操作。如果客户端缓存是陈旧的，定位算法将需要多达 6 次的往返，因为陈旧的缓存值只有在不命中时才会被发现(假设 METADATA tablet 并不会经常移动)。尽管 Tablet 位置信息是存储在内存中(如上所述)，不需要访问 GFS，但是我们还是通过在客户端库里进行预取来降低花费：当读取 METADATA 表时，它会读取不止一个 tablet 的信息。

我们也会将一些额外信息存放在 METADATA 表里，包括对于每个 tablet 有关的事件日志(比如一个服务器何时开始提供服务)。这些信息对于调试和性能分析很有帮助。

5.2 tablet 分配

每个 Tablet 一次只会分配给一个 tablet 服务器。Master 保存了现有的活动的 tablet 服务器集合的所有行踪、tablet 服务器当前分配的 tablet、哪些 tablet 未被分配。当一个 tablet 没有被分配并且有一个可以存储该 tablet 的 tablet 服务器存在时，master 通过给那个 tablet 服务器发送一个 tablet 负载请求来分配该 tablet。

Bigtable 使用 Chubby 来追踪 tablet 服务器的状态。当一个 tablet 服务器启动时，它创建并获取一个在给定的 Chubby 目录上的唯一命名的文件的独占锁。Master 通过监控这个目录(服务器目录)来发现 tablet 服务器。Tablet 服务器如果丢失了它的独占锁(比如由于网络分区导致它丢失了它的 Chubby 会话)就停止它上面的 tablet 服务。(Chubby 提供了一种有效的机制来让一个 tablet 服务器可以检查它是否仍持有它的锁，而不会带来太大的网络开销) tablet 服务器会尝试重新获取在该文件上的独占锁，只要该文件还存在。如果该文件已经不存在了，那么 tablet 服务器就不能再提供服务了，这样它就会自杀{!如果文件已不存在，那么 master 可能已经把它负责管理的那些 tablet 进行了重分配，因此它就不能再继续提高服务，否则可能引起不一致}。当一个 tablet 服务器停止(比如集群管理系统从集群中删除了该 tablet 服务器所在的机器)，它就会尝试释放这个锁，这样 master 就可以尽快地重新分配它上面的 tablets。

Master 负责检测一个 tablet 服务器何时停止提供服务，以尽快重新安排它上面的 tablets。为了进行检测，master 周期性的向每个 tablet 服务器询问它的锁状态。如果一个 tablet 服务器报告它丢失了它的锁，或者 master 在它的几次尝试中不能到达一个 tablet 服务器，master 会尝试获取该服务器的锁。如果 master 可以获取该锁，那么 Chubby 就是活动的，而 tablet 服务器要么是死的要么因为某些问题而无法到达 Chubby，那么 master 就可以通过删除它的 server 文件来使得该 tablet 服务器永远都不能提供服务。一旦一个服务器的文件被删除了，master 就可以将之前分配给该服务器的所有 tablet 移到那些未分配的 tablet 集合中。为了保证一个 Bigtable 集群不会因为 master 和 Chubby 间的网络问题而变得脆弱，如果 master 的 Chubby 会话过期了，master 会自杀{!这样其他的副本就可以有机会尽快成为新的 master}。然而，如前面所述，master 的失败并不会改变 tablet 服务器的 tablet 分配。

当 master 被集群管理系统启动后，在它可以改变 tablets 之前需要知道它们当前的分配状态。Master 在启动时执行如下步骤：1.master 在 Chubby 获得一个唯一的 master 锁，该锁可以防止出现同时生成多个 master 实例。2.master 扫描 Chubby 的服务器目录来找到所有活着的服务器。3.master 与活着的 tablet 服务器通信来发现每个服务器安排了哪些 tablet。4.master 扫描 METADATA 表来找到 tablet 集合。当扫描中碰到一个未被分配的 tablet，master 会将它添加到未分配的 tablet 集合，并对这个 tablet 进行分配。

在 METADATA 的 tablets 未被分配之前，对于 METADATA 的扫描不能进行。因此在开始扫描之前(步骤 4)，如果在步骤 3 没有发现对于 root tablet 的分配，master 会将 root tablet 添加到未分配 tablets 集合中。这个添加将会使 root tablet 变得可以被分配。因为 root tablet 包含所有 METADATA tablets 的名称，master 当扫描完 root tablet 后就能知道 METADATA 的所有的 tablets。

只有当一个表被创建，现有的两个 tablets 合并为一个，或者一个 tablet 被分裂为两个时，现有的 tablet 集合才会发生变化。Master 能够追踪所有的这些变化，因为除最后一种情况外，其他都是 master 负责的。Tablet 分割需要特殊对待，因为它是由一个 tablet 服务器启动的。Tablet 服务器通过将新的 tablet 的信息记录到 METADATA 表中来提交这个分割。当分割提交后，它会通知 master。为了防止分割通知丢失(因为 tablet 服务器或者 master 死了)，当 master 向 tablet 服务器请求加载{!无论是 tablet 服务器或者 master 死了，这个发生分裂的 tablet 都会有一个重新加载的过程}刚刚发生分割的那个 tablet 时，它会检测到这个新的 tablet。Tablet 服务器会将这个分割通知 master，因为它在 METADATA 表中的 tablet 键值仅包含了 master 让它加载的那个 tablet 的一部分。{!假设 master 没有收到这个分割通知，那么它所记录的 tablet 与 METADATA 表中的就是不一致的，这样在它让 tablet 服务器加载该 tablet 时就会发现该不一致}

5.3 Tablet 服务

Tablet 的持久性状态是通过 GFS 进行存储的，如图 5 所示。

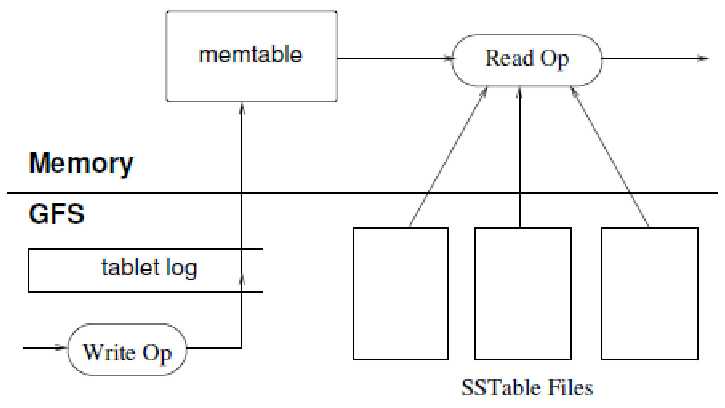


Figure 5: Tablet Representation

更新会提交到一个保存了 redo 记录的提交日志里。在这些更新里，最近提交的那些被保存到内存中一个叫做 memtable 的有序缓存里，老的更新则被保存在一系列的 SSTable 中。为了恢复一个 tablet，tablet 服务器从 METADATA 表中读取该 tablet 的元数据。元数据中包含组成该 tablet 的 SSTable 列表，以及一系列的 redo 点(指向那些包含该 tablet 数据的 commit 日志条目)的集合。Tablet 服务器将所有 SSTable 的索引读入内存，然后通过 replay 那些从 redo 点开始已经提交的更新操作来重新构建 memtable。

{!更新操作在反映到 memtable 中时，会先被保存到 commit log 里，但是当某个服务器挂掉时，它那些保存在 memtable 中最新的更新就不存在了，而 redo 点应该就是记录已保存到 SSTable 的与还在 memtable 中的操作的分界点，这样通过重新执行它之后的那些操作就可以将 memtable 重建}{redo 点何时被更新？有多少个 commit log？参见第 6 节}

当一个写操作到达一个 tablet 服务器时，服务器首先检查它的格式是否合法，发送者是否有权限进行该操作。权限检查是通过从 Chubby 读取一个允许的写操作者列表(通常它直接存在于 Chubby 的客户端缓存中)完成的。一个合法的变更会被写入到已提交日志里。操作的提交可以通过分组执行来提高大量小变更操作出现时的吞吐率，当该写操作提交后，它的内容会被插入到 memtable 里。

当一个读操作到达一个 tablet 服务器时，类似的首先要进行格式和权限检查。一个合法的读操作将会在一系列 SSTable 和 memtable 的一个合并视图上执行{!因为 SSTable 一旦写入就不可变，这样就使得更新操作必须写到新的 SSTable 中，这样就导致同一个 key 值可能在多个 SSTable 中出现，这样读取时就必须读取多个 SSTable 进行 merge 才能得到它真实的最终状态}。因为 SSTable 和 memtable 都是字典有序的数据结构，因此可以很快生成这个视图。{!为了读取一个 key 时，要读入所有的 SSTable，所以第 6 节有一个针对该问题的优化：Bloom Filter。此外伴随着 SSTable 的增多，这种视图合并也会变得低效，所以也引出了下面的 Compation}

当 tablet 发生分割或者合并时，也可以继续接受读写操作。

5.4 compaction

伴随着写操作的执行，memtable 的大小会逐渐变大。当 memtable 大小增长到一个阈值，这个 memtable 就会被冻结，一个新的 memtable 被创建，被冻结的旧的 memtable 会被转化为一个 SSTable 写入 GFS，该过程称为：minor compaction。该过程有两个目的：降低 tablet server 的内存使用，降低该 tablet 服务器挂掉时需要从已提交日志中读取的数据大小。当 compaction 发生时，也可以继续接受读写操作。

每次 minor Compation 会创建一个新的 SSTable。如果这个行为无限制地持续进行，那么读操作就可能会需要从大量的 SSTable 中合并它们的更新。我们通过周期性的执行一个 merging compaction 来将这样的文件数目限制在一定范围内。一个 merging compaction 读取多个 SStable 和 memtable，然后写入到一个新的 SSTable(形成一个最终的归并视图)。一旦这个 compaction 结束，这些 SSTable 和 memtable 就可以丢弃了。

将多个 SSTable 重新写入到一个 SSTable 的 merging compaction 称为主 compaction。由非主 compaction 产生的 SSTable 里可能包含一些在旧的 SSTable 中仍然存活但是目前已经被删除的数据。另一方面，一个主 compaction 产生的 SSTable 不会包含删除操作信息或者已删除数据。Bigtable 循环扫描它所有的 tablet，周期的对它们执行主 compaction。这些主 compaction 使得 Bigtable 可以回收那些被已删除的数据使用的资源。这也保证那些已经删除的数据在一定时间内会从系统中消失，这对于那些存储敏感数据的服务来说很重要。

6 技巧

前面一节描述的实现需要大量的技巧来到达用户所需要的高性能、可用性、可靠性。这一节更细节地描述下实现的各个部分，着重讲述下使用的这些技术：

Locality groups (对应一个 SSTable)

用户可以将多个列族组织为一个 Locality group。对于每个 tablet 里的每个 Locality group 都会生成一个单独的 SSTable。将那些经常不被访问的列族分离到一个独立的群组可以增加访问的效率。比如，webtable 的关于网页的元数据(比如语言，校验和)可放到一个 Locality group，网页内容可以放到另一个群组里，这样一个访问元数据的应用程序就不需要读取所有网页的内容。

另外，一些有用的 tuning 参数也可以以 Locality group 为单位进行设置。比如一个 Locality group 可以声明为放入内存的。对于声明为放入内存的 Locality group 的 SSTable 在需要时才会加载到 tablet 服务器的内存中。一旦加载之后，对于该 Locality group 的访问就不需要访问磁盘。这个特点对于那些需要经常访问的小片数据很有用：比如在 Bigtable 内部我们将它应用在 METADATA 表的 location 列族上。

压缩

用户可以控制对于一个 Locality group 的 SSTables 是否进行压缩，以及使用哪种压缩格式。用户指定的压缩格式会应用在 SSTable 的每个块上(块大小可以通过一个 Locality group 的参数进行控制)。对于每个块单独进行压缩，尽管这使我们丢失了一些空间，但是这使得我们不需要解压整个文件就可以读取 SSTable 的部分内容。很多用户使用一个两遍压缩模式，第一遍压缩使用 Bentley and McIlroy 模式，该模式在一个很大的窗口大小里压缩普通的长字符串。第二遍压缩使用了一个快速压缩算法，该算法在一个小的 16KB 窗口大小内查找重复。这两遍压缩都很快速，压缩速率在 100-200MB/s，解压速率在 400-1000MB/s。

尽管在选择压缩算法时，我们更重视速率而不是空间的减少，但是这个两遍压缩模式工作的出奇地好。比如，在 webtable 里我们使用这种压缩模式存储网页内容。实验中，我们在一个压缩的 Locality group 里存储了大量文档。为了实验目的，我们将文档的版本数限制为 1。该压缩模式达到了 10 :1 的压缩率。这比通常的Gzip 对于HTML网页的3:1或4:1的压缩率要好多了。这是由我们的webtable的行分布方式造成的：来自相同主机的网页被存储在相邻的位置。这使 Bentley and McIlroy 算法可以识别出来自相同站点的大量固有模式。很多应用程序，不仅仅是 webtable，选择的行名称使得类似数据会聚集在一起，因此达到了很好的压缩率。当我们在 Bigtable 中存储相同值的多个版本时压缩率会更好。

为了读性能进行缓存

为了提高读性能，tablet 服务器使用一个两级的缓存机制。Scan Cache 是一个用来缓存由 SSTable 接口返回给 tablet 服务器代码的 key-value 对的高级缓存。块缓存是用来缓存从 GFS 读取的 SSTable 块的低级缓存。Scan Cache 主要用于那些倾向于重复读取相同数据的应用，块缓存则用于那些倾向于从最近读取的数据的邻近位置读取数据的应用(比如顺序读，或者读取一个热点行内的相同 Locality group 里的不同列值)。

Bloom Filters

正如 5.3 节所描述的，一个读操作需要从组成该 tablet 的所有 SSTable 里读取。如果这些 SSTable 不在内存，就需要很多磁盘操作。通过让用户为某个 Locality group 的 SSTables 指定对应的 Bloom filters，可以降低磁盘访问次数。一个 Bloom Filter 允许我们查询对应的 SSTable 是否包含某个给定的 row/column 对的数据。对于特定的应用程序来说，只需要很少的 tablet 服务器内存来保存 Bloom Filter，但可以大大减少读操作所需要的磁盘操作。同时 Bloom Filter 可以避免对于那些不存在的行列的查找访问磁盘。

Commit-log 实现

如果我们为每个 tablet 的提交日志建立一个独立的日志文件，就会使得大量的文

件需要并发写入 GFS。由于每个 GFS 服务器的底层文件系统实现，这些写操作会引起在不同物理日志文件上的大量的磁盘寻道。另外，每个 tablet 一个日志文件会降低分组提交优化的效率。为了解决这些问题，每个 tablet 服务器将更新操作 append 到一个日志文件里，将对于不同的 tablet 的变更放到同一个物理日志文件里。

使用一个日志为正常操作提供了很明显的性能好处，但是使恢复变复杂了。当一个 tablet 服务器挂掉后，它负责的那些 tablet 需要移动到大量其他的 tablet 服务器上：每个服务器通常都会加载一些该服务器的 tablet。为了恢复一个 tablet 的状态，新的 tablet 服务器需要通过原来那个 tablet 服务器的提交日志重新应用这个 tablet 的变更操作。然而对于这些 tablet 的变更是混在同一个日志文件里的。一种方法是，每个新的 tablet 服务器全部读取这个日志文件，然后仅应用那些它需要恢复的 tablet 的变更操作。然而，在这种模式下，如果失败的那台 tablet 服务器的 tablet 被分配到了 100 个机器，那么这个日志文件就需要读取 100 次。

我们通过对提交日志里的 entry 根据<table,row name,log sequence number>进行排序避免了重复的日志读取。在已排序的输出中，对于一个 tablet 的所有变更都是连续的，因此可以通过一次的磁盘寻道和顺序读操作就可以完成读取。为了并行化排序，我们将该日志文件划分为 64MB 大小的段，在不同的 tablet 服务器上对它们进行排序。排序过程是由 master 协调进行的，当一个 tablet 服务器指出它需要从一个日志文件中恢复变更时开始启动。

将提交日志写入 GFS，有时候可能因为各种原因导致性能抖动(比如 GFS 服务器处在繁重的写操作中，或者网络处于拥塞或者重载)。为了避免变更操作受到 GFS 延迟的影响，每个 tablet 服务器实际上有 2 个写日志线程，每个写它们各自的日志文件，在同一时间只有一个处在活跃期。如果对于活动日志文件的写性能急剧下降，它就会切换到另一个线程，在提交日志队列中的那些变更操作将由这个新的活动线程负责写入{!因为日志是写在 GFS 上的，根据 GFS 的原理，不同的文件会由不同的 chunkserver 负责写入，这样当某个文件写入比较慢的时候，如果是因为给它服务的那些 chunkserver 比较忙，这样切换到另一个文件就是可行的}。日志条目里包含序列号，这就使得恢复过程中可以删除那些由于日志切换过程造成的重复条目。

加速 tablet 恢复

如果 master 将 tablet 从一个 tablet 服务器移动到另一个，源 tablet 服务器会首先在该 tablet 上进行一个 minor compaction。这个 compaction 将会减少在 tablet 服务器的日志里的 uncompacted 状态数。当 compaction 结束后，tablet 服务器停止针对该 tablet 的服务。在彻底卸载该 tablet 之前，tablet 服务器再进行一次 minor compaction(通常是很快速的)来消除那些上次 minor compaction 之后该 tablet 服务上剩余的 uncompacted 状态。当第二次的 minor compaction 结束后，该 tablet 就可以直接由另一个 tablet 服务器加载而不需要从日志条目中进行恢复。{ !通过这个过程也可以看出，tablet 服务器只负责管理 memtable 和 SSTable，对于底层的存储它并不负责，当 tablet 迁移到另一个服务器时，它在 GFS 的存储并没有变，

变的只是管理它的 tablet 服务器，而新的 tablet 服务器也不需要进行数据移动之类的操作，因为它同样可以看到原来的 GFS 文件。}

利用不可变性

除了 SSTable 缓存，Bigtable 系统的各部分通过利用"SSTable 生成之后就是不可变的"这个事实也得到了大大的简化。比如我们从 SSTable 中读取时，不需要对文件系统的访问进行任何同步。这样，在行上的并发控制就可以有效的实现。唯一的可以读写的可变数据结构就是 memtable。为了减少在 memtable 读取时的竞争，我们对每个 memtable 行进行写时复制，这就允许读写并行处理。

因为 SSTable 是不可变的，已删除数据的清除就转换成了对于过时的 SSTable 的垃圾回收。每个 tablet 的 SSTables 会注册在 METADATA 表中。Master 服务器采用"标记-删除"的垃圾回收方式删除 SSTable 集合中废弃的 SSTable。

最后，SSTable 的不可变性使得我们可以快速分割 tablet。我们让子 tablets 共享父 tablet 的 SSTables，而不是为每个子 tablet 生成新的 SSTables。{!如果是这样的话，如前所述，一开始只有一个 tablet，这样会不会导致 SSTable 的数目一直未变，只是它的大小一直在上升，但这样会导致它很难一次加载入内存，那么 SSTable 的分割又是何时发生的呢？实际上该处应该是指为了加速分裂过程，SSTable 没必要立即重新生成，而且可以在后面进行主 compaction 的时候去生成它}

7. 性能评价

我们建立了一个 N 个 tablet 服务器的 Bigtable 集群来测量 Bigtable 伴随着 N 变化的性能和可扩展性。每个 Tablet 服务器配置了 1G 内存，它们向一个由配置了 400G IDE 硬盘的 1786 台机器组成的 GFS 单元写入。N 个客户端为这些测试生成工作负载。(我们使用与 tablet 服务器相同数目的客户端来保证客户端不会成为测试瓶颈)。每台机器有一个双核 Opteron 2GHz 芯片，一个 gigabit 以太网链路，以及供运行进程使用的足够的物理内存。机器通过一个两级树状交换机网络连接，根节点总体带宽接近 100-200Gbps。所有机器具有相同的机器配置，同时任意两个机器间的往返时间小于 1ms。

Tablet 服务器和 master，测试客户端，GFS 服务器都运行在相同的机器集合上。每个机器运行一个 GFS 服务器。另外这些机器要么运行一个 tablet 服务器要么运行一个客户端进程，或者一些其他同时使用这些机器的任务的进程。

R 是测试集中 Bigtable 行关键字的个数，它的选值保证每个 tablet 服务器每个基准测试会读写接近 1G 的数据。

顺序写基准测试使用 0-R-1 作为行关键字。这个行关键字空间又被划分为相同大小的 10N 个相同大小的区间。这些区间通过一个中央调度器分配给 N 个客户端，

当客户端处理完分配给它的前一个区间后就继续分配给它的下一个区间处理{!分配是动态的，中央调度器维护一个未分配集合，当发现某个客户端完成后，就给它下一个区间，而不是每个客户端一开始就分配了 10 个固定区间}。这种动态分配方式有助于降低客户端机器上的其他进程造成的性能波动。在每个行关键字下我们写一个字符串，字符串是随机产生因此是未压缩的{!因为是随机产生因此重复的内容会比较少，压缩效果基本无效}。另外，不同行关键字下的字符串也是很不同的，因此跨行的压缩也是不可能的。随机写基准测试与之类似，除了在写之前行关键字是经过 hash 然后模 R 得到的，这样对于整个测试过程来说，写负载就可以在整个行空间上随机分布。

顺序读基准测试与写采用了完全相同的行关键字生成方式。但是它不是在一个行关键字下写，而是读该行关键字下存储的字符串(由前面调用的顺序写基准测试写的)。类似的，随机读基准测试的操作对应着随机写基准测试操作。

扫描基准测试类似于顺序读基准测试，但是它使用了 Bigtable API 对于扫描一个行组的所有值的操作支持。通过使用扫描操作，可以降低基准测试程序所执行的 RPC 调用次数，因为此时的一次 RPC 会从一个 tablet 服务器上获取一大串的值。

随机读(random reads (mem))基准测试类似于随机读(random read)基准测试，但是包含基准测试数据的 locality group 是被标记为 in-memory 的。因此读操作只需要与 tablet 服务器内存交互而不需要读 GFS。对于这个基准测试，我们将每个 tablet 服务器的数据从 1GB 降低到了 100MB,这样它就可以很容易地放到 tablet 服务器的内存里。

图 6 展示了当我们从 Bigtable 读写 1000 字节的 value 值时的测试程序性能。其中表格展示了每个 tablet 服务器的每秒操作数；图形展示了每秒的操作总数。

	# of Tablet Servers			
Experiment	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

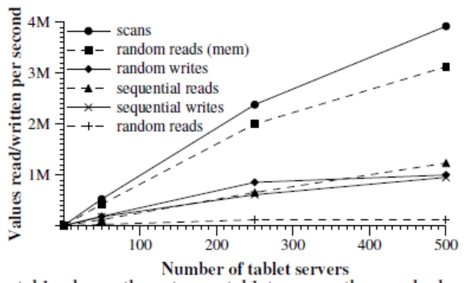


Figure 6: Number of 1000-byte values read/written per second. The table shows the rate per tablet server; the graph shows the aggregate rate.

单 tablet 服务器性能

我们首先来考虑单 tablet 服务器性能。随机读要比所有其他操作慢。每个随机读操作需要将 64KB 大小的 SSTable 块从 GFS 传输到 tablet 服务器，在这些数据里只有 1000 字节会被使用。Tablet 服务器每秒大概执行 1200 次读操作，从 GFS 传输的传输速率大概 75MB/s。在这个带宽级别下，由于网络协议栈、SSTable 解析、Bigtable 代码产生的耗费会占掉大量的服务器的 CPU，同时也几乎占满了我们系统的带宽。大部分具有这种访问模式的 Bigtable 应用需要将块大小设成一个更小

的值，通常是 8KB。

从内存中的随机读要更快些，因为这 1000 个字节的读是从 tablet 的本地内存中直接读取的，不需要从 GFS 中获取一个大的 64KB 块。

随机和顺序写比随机读的执行效率更高是因为每个 tablet 服务器将所有的写入请求写到一个 commit log 里，然后使用按组提交来有效的将这些写入交给 GFS。在随机写和顺序写之间并没有明显的不同；在这两种情况下，对于 tablet 服务器的所有写入都是记录在同一个 commit log 里。

顺序读比随机读执行的更好，因为从 GFS 获取的 64KB SSTable 块会被存储到我们的块缓存里，用来为下面的 64 个请求服务。

扫描操作甚至更快一些，因为 tablet 服务器可以在客户端的一次 RPC 请求中返回更大量的 value，因此 RPC 的耗费可以平摊到大量的 value 上。

扩展性

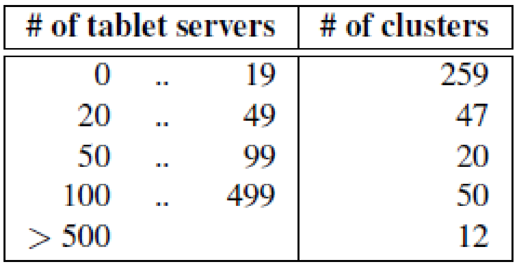
当我们将 tablet 服务器的数目从 1 增长到 500 时，整体的吞吐率有上百倍的增长。比如从内存中随机读的性能当 tablet 服务器数目增长了 500 倍时增长了 300 倍。发生这种情况是因为对于这个基准测试的性能瓶颈在 tablet 服务器 CPU 数。

然而，性能也不是线性增长的。对于大多数基准测试来说，当从 1 增加到 50 个 tablet 服务器时，单服务器的吞吐率有一个明显的下降。这个下降是由于多个服务器时的负载不均衡导致的，通常是由于需要网络和 CPU 的进程引起的。我们的负载平衡算法会尝试解决这种问题，但是无法完美的完成。主要有两个原因：为了减少 tablet 的移动重新的平衡会被抑制(当 tablet 移动时，它会在短时间内不可用，通常小于 1 秒)，随着测试程序的执行它生成的负载也在变化。

随机读基准测试表现出最糟糕的可扩展性(当服务器数目增长了 500 倍后，它只增长了 100 倍)。发生这种情况的原因(正如前面所述)是对于每个 1000 字节的读操作我们都需要在网络上传输 64KB 大小的一个块。这个传输会消耗掉我们网络的 1 GB 共享带宽，这样当我们增加机器数目的时候吞吐率就会明显降低。

8. 实际应用

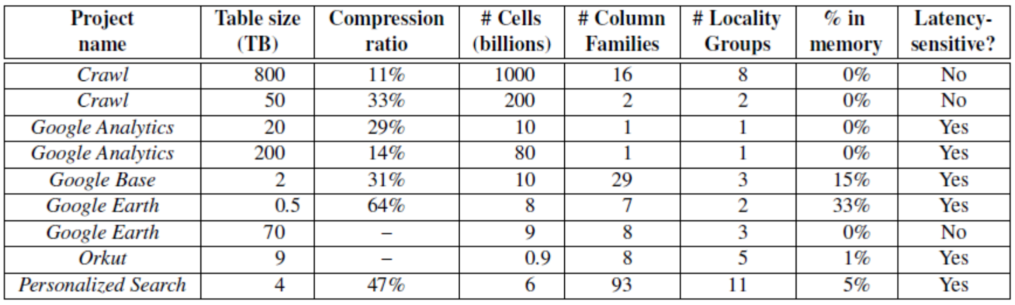
截至 2006 年 8 月，有 388 个非测试性的 Bigtable 集群运行在 Google 机器群上，总共大概有 24500 个 tablet 服务器。表 1 显示了每个集群上的 tablet 服务器个数的粗略分布情况。这些集群大部分是用于开发目的，因此很多时候可能都是空闲的。在 14 个比较忙的集群中，总共有 8069 个 tablet 服务器，每秒有超过 120 万个请求，741MB/s 的 RPC 输入流量，以及 16GB/s 的 RPC 输出流量。



# of tablet servers	# of clusters
0 .. 19	259
20 .. 49	47
50 .. 99	20
100 .. 499	50
> 500	12

Table 1: Distribution of number of tablet servers in Bigtable clusters.

表 2 提供了一些关于现在正在使用的表的数据。一些表为用户存储数据，一些为批处理程序存储数据；这些表在总大小、平均 cell 大小、保存在内存中的数据比例、以及表的 schema 上跨度很大。在该节的剩余部分，我们将简要描述 Google 的三个产品如何使用 Bigtable。



Project name	Table size (TB)	Compression ratio	# Cells (billions)	# Column Families	# Locality Groups	% in memory	Latency-sensitive?
Crawl	800	11%	1000	16	8	0%	No
Crawl	50	33%	200	2	2	0%	No
Google Analytics	20	29%	10	1	1	0%	Yes
Google Analytics	200	14%	80	1	1	0%	Yes
Google Base	2	31%	10	29	3	15%	Yes
Google Earth	0.5	64%	8	7	2	33%	Yes
Google Earth	70	–	9	8	3	0%	No
Orkut	9	–	0.9	8	5	1%	Yes
Personalized Search	4	47%	6	93	11	5%	Yes

Table 2: Characteristics of a few tables in production use. *Table size* (measured before compression) and *# Cells* indicate approximate sizes. *Compression ratio* is not given for tables that have compression disabled.

8.1GoogleAnalytics

Google Analytics 是一个帮助站长分析他们站点的流量模式的服务。它提供整体的统计，比如每天内的不同访问者数目，每个 URL 每天的访问数，以及一些站点反馈报告，比如那些打开了某特定页面后的访问者进行了交易的比例。

为了使用这项服务，站长需要在他们的页面里嵌入一个小 javascript 程序。它会将关于请求的各项信息记录在 Google Analytics 里，比如用户标识以及该网页被获取的信息。Google Analytics 对这些数据进行分析并给站长使用。

我们简要描述 Google Analytics 使用的两个表。原始的点击表(大概 200TB)为每个终端用户会话维护一条记录。行的名称是一个由站点名称，以及会话创建时间组成的元祖。这种 schema 使得那些访问相同站点的会话是相邻的，而且是按照时间排序的，这个表格可以压缩到原始大小的 14%。

摘要表格(大概 20TB)包含对每个站点各种预定义的摘要。这个表格是通过周期性

调度的 MapReduce jobs 从原始点击表生成出来的。每个 MapReduce job 从原始点击表抽取出最近的会话数据。系统整体的吞吐率由 GFS 的吞吐率决定，这个表格可以压缩为原始大小的 29%。

8.2Google 地球

Google 提供一组服务，使得用户既可以通过网页也可以通过 Google 地球客户端软件访问地球表面的高分辨率卫星图像。这些产品允许用户浏览地球表面：他们可以在不同的分辨率上拍摄，观看，注释卫星图像。系统使用一个表来预处理数据，用另外一些表来为用户提供数据。

预处理流程使用一个表来存储原始图像。在预处理期间，图像被整理，然后 合并为最终的服务数据。这个表大概有 70TB 数据，因此需要通过硬盘提供服务{!无法全部放入内存}。图像本来已进行了压缩，因此 Bigtable 的压缩选项被关掉了。

图像表里的每一行对应一个地理区域，行的命名方式保证相邻的地理位置会被存储到邻近的位置。表格包含一个列族来保存每个区域的数据源。这个列族有大量的列：每个列保存一个原始数据图片。因为每个区域仅仅从是几个图片构建出来的，因此这个列族是很稀疏的。

这个预处理流程依赖于 MapReduce 在 Bigtable 上进行数据转换。在这些 MapReduce jobs 运行期间，整个系统每个 tablet 服务器的数据处理速度超过 1MB/s。

服务系统使用一个表来索引存储在 GFS 中的数据。这个表相对小一些(大概 500GB)，但是每个数据中心每秒必须要为数千个查询提供低延时服务。因此，这个表实际上会占用几百个 tablet 服务器，包含许多 in-memory 列族。

8.3 个性化搜索

个性化搜索是一个用来记录用户在 Google 很多产品(比如网页搜索、图像新闻搜索)的查询和点击记录的可选服务。用户可以通过浏览搜索历史来查看过去的查询和点击，可以通过他们在 Google 的历史记录得到个性化的搜索结果。

个性化搜索将每个用户的数据保存在 Bigtable 里。每个用户具有唯一的用户 id，并分配给他一个以用户 id 命名的行。所有的用户动作保存在表中。每个类型的动作用一个独立的列族保存(比如有一个列族保存所有的网页查询)。每个数据元素使用对应动作的发生时间作为它在 Bigtable 中的时间戳。个性化搜索通过在 Bigtable 上的 MapReduce 产生用户特征。这些用户特征被用来个性化实时搜索结果。

为了提高可用性和降低用户延时，个性化搜索数据备份在多个 Bigtable 集群上。个性化搜索团队起初自己在 Bigtable 之上建立了一个用户端备份机制来保证所有

备份的一致性。现在的系统已经使用一个内建到服务端的备份子系统了。

个性化搜索存储系统的设计允许其他团队在他们的列中添加新的用户信息。现在这个系统已经被很多其他需要存储用户配置和设置信息的 Google 产品使用。多个团队间共享一个表使得表具有大量的列族。为了支持共享，我们给 Bigtable 添加了一个简单的 quota 机制来限制共享表的用户的存储消耗。这个机制为不同产品团队使用该系统进行用户信息存储提供了隔离性。

9. 经验教训

在 Bigtable 的设计、实现、维护和支持过程中，我们获得了很多经验以及一些教训。

我们学到的第一个经验是：大型分布式系统会经常面对很多类型的失败，这些失败不仅仅是标准的网络分区，各种分布式协议中的失败。比如我们看到了由下面的各种原因引起的失败：内存和网络损坏，大的时钟误差，机器挂起，扩展的非对称的网络分划，我们使用的其他系统的 bug(比如 chubby)，GFS quota 溢出，计划的以及临时的硬件维护。通过改变各种协议解决这些问题，我们得到了更多经验。比如，我们为我们的 RPC 机制增加了校验和。我们也通过消除系统中的一部分对另一部分的依赖来处理一些问题。比如我们不再假设 Chubby 只会返回一组固定错误集合中的错误{!可能返回任意错误}。

我们学到的另一个经验是：将新 feature 的添加延迟到已经清楚它会被怎样使用了时是很重要的。比如，一开始我们计划在 API 中提供一个通用的事务支持。因为当时我们没有一个现实的使用需求，所以也没有实现它。现在我们有很多运行在 Bigtable 上的实际应用，我们检查它们的实际需求，发现大多数的应用只需要一个单行事务。在已经提出的分布式事务需求中，最重要的使用是用来维护 secondary 索引，我们计划通过添加一个特殊的机制来满足这个需求。新的机制不如分布式事务通用，但是会更有效(尤其是在数百行或者更多数据上进行更新)也会能够更好地与我们的跨数据中心的备份机制进行交互。

我们在进行 Bigtable 支持中学到的一个比较特殊的经验是：合适的系统级监控的重要性(比如既监控 Bigtable 自身，同时还要监控使用 Bigtable 的用户进程)。比如我们扩展 RPC 系统使得可以追踪一次 RPC 中的重要动作。这个特点帮助我们检测并解决了很多问题比如在 tablet 数据结构上的锁竞争，在提交 Bigtable 变更时的 GFS 上的低速写，当 METADATA tablet 不可用时造成的 METADATA 表不可访问。另一个有用的监控是每个 Bigtable 集群会注册在 Chubby 上。这就使得我们可以追踪所有的集群，发现它们的大小，检查它们使用的软件版本，它们收到的流量，是否存在一些问题比如未预料到的大的延迟。

我们学到的最重要的经验就是简单设计的价值。考虑到我们系统的规模(非测试性代码大概 10 万行)，以及代码会随着时间的推移以不可预料的方式演变，我们发现代码和设计的清晰对于代码的维护和调试有巨大的帮助。一个例子是我们的

tablet 服务器成员协议。我们最初的协议很简单：master 周期性的给 tablet 服务签订租约，当租约过期后 tablet 服务器就会自杀。不幸的是，这个协议很明显地降低了在网络问题出现时的可用性，而且对于 master 的恢复时间也很敏感。我们进行了多次设计才得到一个执行的不错的协议。然而最终的协议变得太复杂而且依赖于 Chubby 系统中那些很少被其他应用所使用的 feature。我们发现我们花费了大量时间调试令人费解的边边角角的问题，不仅仅是 Bigtable 代码，有时还是在 Chubby 代码里。最后，我们放弃了这个协议，使用了一个新的简化的协议，它只依赖于 Chubby 那些被广泛使用的 feature。

10. 相关工作

Boxwood 项目有些组件与 Chubby、GFS、Bigtable 在某些方面重叠，因为它提供了分布式协商，锁，分布式 chunk 存储，分布式 B 树存储。尽管存在这些方面的重叠，但是很明显的是与对应的 Google 服务相比 Boxwood 组件的定位是在更底层上。Boxwood 项目的目标是为建立上层服务比如文件系统或者数据库提供基本的设施，而 Bigtable 的目标则是直接支持那些需要存储数据的用户应用程序。

最近很多项目都是旨在解决提供分布式存储和在广域网上的上层服务的问题，通常是在整个互联网范围内。这包括在分布式 hash 表上的工作，对应的项目有 CAN，Chord，Tapestry 和 Pstry。这些项目关注点与 Bigtable 不同，比如高可用带宽、不可信任成员、频繁重配置、非中央式控制、拜占庭式容错，这些都不是 Bigtable 的目标。

在提供给应用程序开发者的分布式数据存储模型上，我们认为由分布式 B 树或者是分布式 Hash 表提供的 key-value 对模型提供的功能有限。Key-value 对是一种很有用的构建模式，但是它们并不应该是提供给开发者的唯一模式。我们所选择的模型要比简单的 Key-value 模式丰富，支持半结构化数据。然而，它仍然是足够简单地，使得在处理 flat-file 格式时也很有效，同时它也允许用户透明地调整(通过 locality group)系统的重要行为。

一些数据库厂商已经开发出了可以用来存储大量数据的并行数据库。Oracle 的 RAC(Real Application Cluster)数据库使用共享磁盘来存储数据(Bigtable 使用 GFS)和一个分布式锁管理器(Bigtable 使用 Chubby)。IBM 的 DB2 并行版基于类似于 Bigtable 的非共享架构。每个 DB2 服务器负责表中行的一个子集，存储在本地的关系数据库中。这两个产品都提供了支持事务的完全的关系模型。

Bigtable 的 locality group 实现了与其他的基于列而不是基于行的磁盘数据组织系统(包括 C-Store,以及一些商业性产品比如 Sybase IQ、SenSage、KDB+、MonetDB/X100 中的 ColumnBM 存储层)类似的压缩率和磁盘读性能的提高。另一种将垂直和水平数据划分到 Flat 文件，并且达到了很高压缩率的系统是 AT&T 的 Daytona 数据库。locality group 并不支持 Ailamaki 描述的那些 CPU 缓存级的优化。

Bigtable 使用 memtable 和 SSTables 存储对于 tablet 更新的方式类似于

Log-Structured Merge Tree 存储索引数据更新的方式。在这两个系统中，已排序的数据在写到磁盘之前都是要缓存在内存中，读的时候必须从内存和磁盘里 merge 数据。

C-Store 和 Bigtable 有很多共同点：这两个系统都使用了非共享的架构，都有两个数据结构，一个用来保存最近的写，一个用来保存长期存在的数据，存在一个把数据从一个搬到另一个的机制。这两个系统在它们的 API 上存在显著的不同：C-Store 类似于一个关系数据库，而 Bigtable 提供了底层的读写操作接口，而且设计得可以支持每秒每个服务器执行数千次这样的操作。C-Store 也可以说是一个”读优化的关系型 DBMS”，而 Bigtable 则为读敏感和写敏感的应用都提供了好的性能。

Bigtable 的负载平衡器需要解决与非共享数据库碰到的一些相同的负载和内存均衡问题。我们的问题要简单一些：(1)我们不需要考虑相同数据由于索引或者视图产生的多个拷贝的可能性；(2)我们让用户决定数据是放在内存还是磁盘，而不是动态确定；(3)我们没有复杂的查询执行和优化。

11. 总结

我们已经描述了 Bigtable，Google 的一个存储结构化数据的分布式系统。Bigtable 集群自从 2005 年 4 月开始在产品中使用，之后我们又在设计和实现上花费了大概 7 个人年(person years)。截至 2006 年 8 月，有超过 60 个的项目在使用 Bigtable。我们的用户很喜欢我们的 Bigtable 实现提供的性能和高可用性，当随着时间的推移他们的系统资源需求发生变化时，他们可以简单地添加机器来扩展集群的容量。

由于 Bigtable 提供的不一样的接口，有一个有趣的问题是对于用户来说适应它有多困难。新用户有时候不确定如何最好的使用 Bigtable 接口，尤其是如果他们已经习惯于使用支持通用事务的关系数据库。然而，事实是 Google 的很多产品在实践中成功地使用 Bigtable 完成了他们的设计工作。

我们目前正在实现 Bigtable 的几个额外的 feature，比如支持二级索引，具有多 master 备份的跨数据中心的 Bigtable 备份。我们已经开始将 Bigtable 为产品团队部署为一个服务，这样不同的团队就不需要维护他们自己的集群。伴随这服务集群的扩展，我们将需要在 Bigtable 内部处理更多的资源共享问题。

最后，我们发现在 Google 内部部署我们自己的存储系统具有明显的优点。我们从我们为 Bigtable 设计的数据模型中得到了大量的灵活性。另外，我们可以控制 Bigtable 的实现，以及 Bigtable 所依赖的其他基础模块的实现，意味着当瓶颈出现或者效率降低时，我们可以消除它们。

致谢

……

译考文献：
HBase技术介绍 http://www.searchtb.com/2011/01/understanding-hbase.html
HFile存储格式 http://www.tbdata.org/archives/1551

Chubby：面向松散耦合的分布式系统的锁服务

作者：Mike Burrows Google Inc 2006
译者：phylips@bmy 2011-7-2
出处：http://duanple.blog.163.com/blog/static/709717672011264394617 8/

摘要

本文描述了我们在 Chubby 锁服务系统上的相关经验。该系统旨在为松散耦合的分布式系统提供粗粒度的锁以及可靠性存储(低容量的)。Chubby 提供了一个非常类似于具有建议性锁的分布式文件系统的接口，设计的侧重点在可用性及可靠性而不是高性能。该服务已有很多应用多年的运行实例，其中一些实例可能同时为成千上万个客户端提供服务。本论文描述了其最初的设计及期望的应用，并通过将其与实际应用情况进行对比，来说明设计应如何修正以容纳各种差异。

1. 导引

本文描述一个称为 Chubby 的锁服务。它被用在由通过高速网络连接的大量小型计算机组成的松耦合分布式系统中。比如一个 Chubby 实例(也称做一个 Chubby 单元)可能服务于一个由 1Gbit/s 以太网连接的上万台 4 核计算机组成的集群。大多数的 Chubby 单元只是存在于一个计算中心或者一个机房，当然也会有些 Chubby 单元，它们的副本可能分布在相隔数千公里的地方。

提供锁服务的目的是为了允许客户端可以同步它们自己的行为或者在某些基本环境信息上达成一致。首要的设计目标包括可靠性，面对大量客户端集合时的可用性，以及易于理解的语义；而吞吐率和存储能力是一些次要的考虑因素。Chubby 的客户端接口类似于一个支持整文件读写，具有建议性锁及事件(比如文件内容改变)通知机制的简单文件系统。{!建议锁(advisory lock)，这种锁是用于协调工作的一种锁，系统只提供加锁及检测是否加锁的接口，系统本身不会参与锁的协调和控制，可能有用户不进行是否加锁的判断，就修改某项资源，这时系统是不会加以阻拦的。因此这种锁不能阻止用户对互斥资源的访问，只是提供给访问资源的用户们进行协调的一种手段，所以资源的访问控制是交给用户控制的。与此相对的则是强制锁(mandatory lock)，此时，系统会参与锁的控制和协调，用户调用接口获得锁后，如果有用户不遵守锁的约定，系统会阻止这种行为}

我们希望 Chubby 可以帮助开发者处理他们系统中的粗粒度的同步问题，尤其是一类 leader 选举问题，即从一组等价的服务器集合中选择出一个领导者。比如，GFS 使用 Chubby 锁来选出一个 GFS master 服务器，Bigtable 通过如下几种方式使用 Chubby：master 选举，帮助 master 找到它控制的服务器集合，帮助客户端找到 master。此外 GFS 和 Bigtable 都使用 Chubby 存储它们一部分的元数据，更进一步地说，实际上它们是使用 Chubby 作为它们的分布式数据结构的根节点。还

有一些服务使用锁在多个服务器间划分任务(在比较粗的粒度上)。

在部署 Chubby 之前，Google 的大多数分布式系统，使用一种自适应的方法来解决主从选举问题(如果可以进行一些重复性的工作而不会有什么害处)，或者是需要人工干预(如果正确性至关重要)。对于前一种情况，Chubby 可以节省一些计算能力，对于后一种情况，它可以让系统在出现错误时不再需要人的干预。

熟悉分布式计算的读者应该知道，在多个节点中的 leader 选举问题是分布式一致性问题的一个实例，而且应该意识到我们需要一个使用异步通信(该名词描述了大多数实际网络的行为，比如以太网和因特网，它们允许丢包，延时，以及乱序)方式的解决方案。异步一致性问题可以通过 Paxos 协议解决，Oki 和 Liskov 在它们的论文(viewstamped replication)中提出了一个与之等价的协议。实际上，目前我们所看到的所有可工作的异步一致性协议都是以 Paxos 为核心。Paxos 虽然不需要对时钟做任何假设就可以保证安全性，但是还是需要引入时钟以确保活性(即保证程序会成功结束)；这就克服了 Fisher 等提出的 FLP 不可能性结论。

构建一个满足上面所提的各种需求的 Chubby 系统，本质上主要是一种工程性的努力，而算不上是新的研究成果，我们并没有提出新的算法或技术。本文的目的是描述下我们做了什么，以及为何那样做。在下面的章节里，我们会描述下 Chubby 的设计与实现，以及它的演变过程。我们会讲述一些 Chubby 令人意想不到的一些使用方式，以及一些已经证明是错误的特性。我们忽略了一些背景性的细节知识，比如一致性协议或者 RPC 系统的细节。

2. 设计

2.1 原理

有人可能认为我们应该构建一个支持 Paxos 协议的库，而不是提供一个访问中央锁服务的库，即使该服务具有很高的可靠性。一个客户端 Paxos 库不会依赖于其他的服务端(除了命名服务器)，而且也可以为程序员提供一个标准框架(如果他们的服务可以以状态机的形式实现)。实际上，我们也提供了这样的一个独立于 Chubby 的客户端库。

然而，与客户端库相比，锁服务的形式有如下一些优点：首先，开发者有时并不像我们想象的那样会考虑到程序的高可用性，通常他们的系统从一个只有很少负载及可用性保证的原型起步；代码也没有为使用一致性协议而进行一些特殊的结构化设计。伴随着服务的成熟及客户的增多，可用性变得越来越重要，replication 及主从选举也会被加入到现有设计中。虽然这些可以通过一个提供分布式一致性的库来实现，但是锁服务器可以使得维护现有代码结构及通信模式更简单。比如，为了选举一个 master 并将选举结果写入一个文件服务器，只需要在现有系统中增加两个语句及一个 RPC 参数即可：一条语句负责获取锁变成 master，同时传递一个额外的整数(锁的获取计数)给写 RPC 调用，然后为文件服务器添加一个 if 语句，如果该数小于当前值就拒绝该写请求{!获取锁计数的目的是为了防止之前

的过期请求，首先 master 选择是一个持续的过程，也就是说该选举会进行多次，比如当第一次选举结束后，假设节点 a 被选为 master，这样它就应该向文件服务器发起写 RPC 调用，但是该 RPC 调用可能因网络被延迟，这样可能会发起第二次选举，比如第二次 b 被选为了 master，于是呢它也向文件服务器发起写 RPC 调用，在这个 RPC 调用结束之后，a 的调用可能才到达，为了避免 a 的结果覆盖 b 的结果，通过锁的计数我们就能知道这个延迟到达的 RPC 调用不应被执行了}。我们发现这种方式要比为现有的服务器加入一致性协议要简单许多，尤其是在迁移期间还要求保留兼容性的时候。

第二，那些需要进行 leader 选择的服务，通常都需要一种机制将选举结果广而告之。这意味着我们需要允许客户端能够存储和获取少量数据—即需要读写小文件。这可以通过一个命名服务来解决，但是经验告诉我们锁服务本身很适合完成这种任务，这即减少了客户端需要依赖的服务器数，同时也让协议的一致性特性得到了共享。Chubby 可以成功地作为一个命名服务器使用，主要归功于它使用了一个一致性的客户端缓存机制，而不是基于时间的缓存机制。尤其是，我们发现开发者很高兴看到不用再去确定一个类似于 DNS ttl 的一个缓存过期时间参数，而一个糟糕的设置可能导致高 DNS 负载或者过长的客户端故障恢复时间。

第三，基于锁的接口对于程序员来说更熟悉。具有多副本状态机的 Paxos 及带有互斥锁的临界区都可以为程序员提供串行编程的效果。但是，很多程序员以前就接触过锁，而且认为知道如何使用它们，然而讽刺的是，他们通常是错误的，尤其是当他们在分布式系统中使用锁的时候。很少有人会考虑机器失败对异步通信系统的锁带来的影响。虽然使用锁会阻碍程序员为分布式的决策使用可靠机制的思考，但是这种对于锁的熟悉性，还是战胜了这种副作用。

最后，分布式一致性算法使用 quorums 做决策，因此可以使用多个副本来达到高可用性。比如 Chubby 的一个单元通常具有 5 个副本，只要其中的 3 个正常，该单元就可以提供服务。与此同时，如果客户端系统使用锁服务，即使只有一个客户端可以获取锁就可以保证安全的往前推进。因此，锁服务降低了保证客户端系统可以正常进展的所需的服务器数目。更宽泛地说，可以认为锁服务提供了一种通用的选举机制，它允许客户端系统在其自身成员存活数小于半数时仍可以正确地做出决策{!如果采用客户端库，这样就不存在一个服务集合，而是依赖于客户端本身集合进行决策，这样在客户端系统存活数少于半数时，就无法进行决策了}。有人可能想，可以通过另一种方式解决这个问题：通过提供一个”一致性服务”，使用一组服务器组成 Paxos 协议中的 acceptors 集合。与锁服务类似，”一致性服务”也需要保证在即使只有一个客户端存活的情况下也能保证安全的向前推进；类似的技术已经被用来减少拜占庭容错情况下所需的状态机数目。然而，如果”一致性服务”不提供锁机制，那么它就没办法解决上面所提过的其他问题。

上面这些观点可以推出两个关键的设计决定：

- 我们选择了锁服务的形式，而不是客户端库或者”一致性服务”的形式
- 我们提供小文件来允许被选定的 primaries 来公布它们自身及一些参数，而不是再去创建和维护另一个服务

还有一些决定来自于我们所期望的应用及环境：

- 一个通过 Chubby 文件来发布其 primary 的服务可能有成千上万个客户端，因此，我们必须允许这些客户端能够查看这个文件，又不能需要太多的服务器。
- 客户端和一个具有备份机制的服务的副本们可能希望能够获知服务的 primary 的改变。这意味着需要使用一种事件通知机制避免轮询。
- 即使客户端不会周期性地去轮询文件，但是由于系统支持多个开发者也会导致文件的访问很频繁，因此文件的缓存是必要的。
- 开发者可能会被非直观的缓存语义搞糊涂，因此我们需要一致性缓存。
- 为了避免金钱上的损失及牢狱之灾，我们需要提供包括访问控制在内的一些安全机制。

一个可能让读者感到惊讶的选择是我们的锁并不适用于细粒度的应用场景，在该情况下，锁可能仅仅是在一个非常短的时间(秒级甚至更短)被持有，事实上它是为粗粒度的应用而诞生的。比如，一个应用可能使用锁来选举一个 primary，该 primary 会在相当长的时间内处理所有的数据访问请求，可能是数小时或者数天。这两种不同风格的使用方式，对锁服务器提出了不同的需求。

粗粒度的锁带给锁服务器的负载很低。尤其是，锁的获取率与客户端应用程序的事务发生率通常只是弱相关的。粗粒度的锁很少产生获取需求，这样锁服务偶尔的不可用也很少会影响到客户端。另一方面，锁在客户端之间的传递可能会需要昂贵的恢复过程，这样我们就不希望锁服务器的故障恢复会导致锁的丢失。因此，最好在锁服务器出错时，能让粗粒度的锁仍然有效，不需要考虑这样做带来的开销，而且这样就允许由一些较低可用性的锁服务器就足以为很多客户端提供服务。

细粒度的锁会导致完全不同的结论。即使锁服务器在简短的时间内不可用，也可能导致很多客户端的失败。性能以及随意增加锁服务器的需求就需要着重考虑，因为锁服务器端的事务发生频率与客户端的事务发生频率完全相关联。当然它也有一些优点，比如由于不需要维护在锁服务器失败时的锁状态而可以降低锁的开销，而且由于锁的持有时间通常都很短，这样由频繁丢锁所造成的时间惩罚也不会那么严重了。客户端的设计必须是已经考虑到网络分区发生期间的锁丢失，这样就不需要在锁服务器故障恢复过程中去恢复它们。(Clients must be prepared to lose locks during network partitions, so the loss of locks on lock server fail-over introduces no new recovery paths)。

Chubby 只是提供粗粒度的锁。幸运的是，客户端根据自己的应用特点量身定制细粒度的锁是很简单的。一个应用可以将它的锁划分成多个组，然后使用 Chubby 的粗粒度锁服务为这些组分配一个应用级的锁服务器。为维护这些细粒度的锁只需要记录很少的状态，服务器只需要记录一个很少更新、非易失、严格递增的获取计数器。(Clients can learn of lost locks at unlock time)客户端在解锁的时候能够知道丢失的锁{?何意？}，如果 使用一个简单的定长租约机制，协议就能变得简单而有效。这种模式带来的最重要的好处是，客户端开发者现在可以自己提供那些支持它们自己的负载的应用级锁服务器，而且也简化了自己去实现一个一致性协议的复杂性。

2.2 系统结构

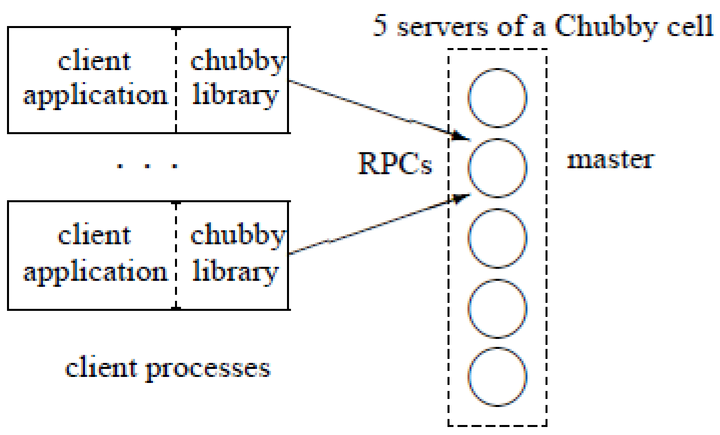


Figure 1: System structure

Chubby 有两个主要组件，它们通过 RPC 通信：一个服务器，一个客户端应用程序需要链接的库，如图 1 所示。客户端与服务端之间的所有通信都需要通过客户端库。还有一个可选组件：代理服务器，将在 3.1 节讨论。

一个 Chubby 单元由被称为副本的服务器(通常是 5 个)集合组成，同时采用特殊的放置策略以尽量降低关联失败的可能性(比如它们通常在不同的机柜中)。这些副本使用分布式一致性协议来选择一个 master；master 必须获得来自副本集合中的半数以上的选票，同时需要保证这些副本在给定的一段时间内(即 master 的租约有效期间内)不会再选举出另一个 master。Master 的租约会被周期性地更新只要它能够持续获得半数以上的选票。

每个副本维护一个简单数据库的一个拷贝，但是只有 master 会读写该数据库，所有其他的副本只是简单地复制 master 通过一致性协议传送的更新。

客户端通过通过向 DNS 中列出的各副本发送 master 定位请求来找到 master。非 master 副本通过返回 master 标识符来响应这种请求。一旦客户端定位到 master，它就会将自己的所有请求直接发送给 master，直到要么它停止响应，要么它不再是 master。写请求会通过一致性协议传送给所有副本，当写请求被一个 Chubby 单元中半数以上的副本收到后，就可以认为已成功完成。读请求只能通过 master 处理，只要 master 租约还未到期这就是安全的，因为此时不可能有其他 master 存在。如果一个 master 出错后，其他的副本就可以在它们的 master 租约过期后运行选举协议，通常几秒钟后就能选举出一个新的 master。比如最近的两次选举花了 6s 和 4s，当然我们也曾看到过这个时间有时会高达 30s。

如果副本出错而且几个小时内都无法恢复，一个简单的替换系统会从一个空闲机器池内选择一个新机器来，然后在它上面运行锁服务器的二进制文件。然后它会更新 DNS 表，将出错的那台机器对应的 IP 地址更新为新的。当前的 master 会周期性地去检查 DNS 表，最终会发现该变化。然后它就会更新它所在的 Chubby 单

元的成员列表，该列表通过普通的复制协议来维持在多个副本间的一致性。与此同时，这个新的副本会从存储在文件服务器上一组备份中选择一个数据库的最近的拷贝，同时从活动的那些副本中获取更新。一旦该新副本已经处理过当前 master 正在等待提交的请求后，该副本就被允许在新 master 的选举中投票了。

2.3 文件、目录和句柄

Chubby 提供了一个类似于 UNIX 但是相对简单的文件系统接口。它由一系列文件和目录所组成的严格树状结构组成，不同的名字单元之间通过反斜杠分割。一个典型的名称如下：
/ls/foo/wombat/pouch
对于所有的 Chubby 单元来说，都有一个相同的前缀 ls。第二个名字单元(foo)代表了 Chubby 单元的名称；通过 DNS，它会被解析成一个或多个 Chubby 服务器。一个特殊的单元名称 local，用于指定使用客户端本地的那个 Chubby 单元；通常来说这个本地单元都与客户端处于同一栋建筑里，因此也是最可能被访问的那个。剩下的名字单元/ wombat/pouch，将会由指定的那个 Chubby 单元自己进行解析。与 UNIX 类似，每个目录由一系列的子文件和目录组成，每个文件包含一系列字节串。

因为 Chubby 的名字空间结构类似于文件系统，这样就使得可以为应用提供我们自己特定的 API，也可以使用我们的其他文件系统的接口，比如 GFS。这就显著降低了我们为编写名字空间浏览及操纵工具所需要付出的努力，同时也降低了培训 Chubby 用户的难度。

与 UNIX 不同的是，该设计旨在简化分布式。为允许不同目录下的文件由不同的 Chubby master 负责，我们禁止了文件和目录的 mv 操作，不再维护目录修改时间，同时避免了那些依赖于路径的权限语义(即文件访问是由文件本身的访问权限控制的，与其父目录无关)。为了更容易缓存文件元数据，系统也不再提供最后访问时间。

名字空间由文件和目录组成，统称为 node。每个 node 在一个 Chubby 单元中只有一个名称与之关联；不存在符号连接或者硬连接。node 要么是永久性的要么是临时的。所有的 node 都可以被显示地删除，但是临时节点在没有 client 端打开它们(对于目录来说，则是为空)的时候也会被删除。临时节点可以被用作中间文件，或者作为 client 是否存活的指示器。任何节点都可以作为建议性的读/写锁；关于锁的进一步的细节参考 2.4 节。

每个节点都包含一些元数据。包括三个访问控制列表(ACLs)，用于控制读、写操作及修改节点的访问控制列表(ACL)。除非显式覆盖，否则节点在创建时会继承它父目录的访问控制列表。访问控制列表(ACLs)本身单独存放在一个特定的 ACL 目录下，该目录是 Chubby 单元本地名字空间的一部分。这些 ACL 文件由一些名字组成的简单列表构成，说到这里，读者可能会联想到 Plan 9 的 groups。因此，如果文件 F 的写操作对应的 ACL 文件名称是 foo，那么 ACL 目录下就会有一个文件 foo，同时如果该文件内包含一个值 bar，那就意味着允许用户 bar 写文件 F。

用户通过一种内建于 RPC 系统的机制进行权限认证。Chubby 的 ACLs 就是简单的文件，因此对于其他想使用类似的访问控制机制的服务可以直接使用它们。

每个节点的元数据还包含 4 个严格递增的 64 位数字，通过它们客户端可以很方便的检测出变化：

- 一个实例编号；它的值大于该节点之前的任何实例编号
- 一个内容世代号(只有文件才有)；当文件内容改变时，它的值也随之增加
- 一个锁世代号；当节点的锁从 free 变为 hold 时，它的值会增加
- 一个 ACL 世代号；当节点的 ACL 名字列表被修改时，它的值会增加

Chubby 也提供一个 64 位文件内容校验和，这样 client 就可以判断文件内容是否改变了。

Client 通过 open 一个节点获取 handle(类似于 UNIX 的文件描述符)。handle 会包括如下一些东西：

- 检查位，用于防止 client 端伪造或者猜测 handle，这样完整的访问控制检查只需要在 handle 被创建出来的时候执行即可(与 UNIX 相比，它只是在 open 的时候检查权限位，而不是每次读写都去检查，因为文件描述符是不可伪造的){!handle 的创建与 open 是两个概念？根据 2.6 节可以知道句柄只会在 open 时创建。是说 UNIX 还是说 Chubby 每次读写都需要进行权限检查？应该是说 chubby 不需要每次读写都检查权限，因为有了检查位，它只需要检查这个检查位即可。同时参考 linux 的 read 和 write 实现，在 vfs_read 中可以看到，每次调用它都会做如下检查 if (!(file->;f_mode & FMODE_READ))，而在 vfs_write 里则有 if (!(file->;f_mode & FMODE_WRITE))，此外二者还都会调用 security_file_permission ()做进一步的检查。}
- 一个序列号，使得 master 可以判断该 handle 是由它还是之前的那个 master 生成的。
- 在 open 时所提供的模式信息，允许重启后的 master 可以重建一个被转移给它的旧 handle 的状态。

2.4 锁和序列号

每个 Chubby 文件和目录都可以作为一个读者-写者锁：要么是一个 client 以独占(writer)模式持有它，要么是任意数量的 client 以共享(reader)模式持有它。类似于我们所熟知的 mutex，锁是建议性的。也就是说，当多个 client 同时尝试获得相同的锁时，它们会产生冲突：但是持有 F 的锁，既不是访问文件 F 的必要条件，也不能阻止其他 client 的访问。我们没有使用强制性锁(它会使得对于那些没有拿到锁的 client 无法访问被锁住的对象)：

- Chubby 锁经常被用来保护其他服务的资源，而不仅是与锁关联的那个文件。要让强制性的锁真正有意义，还需要我们对这些服务本身做更多的修改才行。
- 我们不希望当用户因 debug 或者管理的需要而访问那些被锁住的文件时，必须要关闭应用程序才行。在一个复杂系统中，很难采用那种在 pc 上经常使用的策略，比如在 pc 上，系统管理软件可以简单地通过让用户关闭或者重启应用程序，就可以打破强制性锁。

- 我们的开发者通过很方便的方式就可以进行一些错误检查，比如通过写出一个诸如”lock X is held”的断言，因此他们从强制性锁中很少获益。而那些没有获得锁的恶意进程，有很多方式和机会去破坏数据，因此由强制性锁所提供的这种额外保证意义就更微弱了。

在 Chubby 里，获取任何模式的锁都需要写权限，因此一个无权限的读者无法阻止一个写者的操作{!为何 reader 无法阻止一个 writer？首选因为锁是建议性的，因此一个读者可以不申请锁就去读，这样它就根本无法阻止其他人同时去写，即便是其他人在操作时首先尝试去获取锁，但是因为读者没有持有锁，这样其他访问者根本不知道有人在读}

在分布式系统中，锁是很复杂的，因为通信通常是不确定的，进程可能会 fail，而相互之间却毫无所知。比如，一个持有锁 L 的进程可能发出一个请求 R，然后 fail 了。另一个进程可能获得了锁 L，同时在 R 到达目的地之前执行了一些操作。之后，R 又到达了，这样它就可能在没有锁 L 的保护下进行一些操作，这样就可能产生一些不一致的数据。关于消息的乱序到达问题已经被研究了很多了；一些解决方案比如 virtual time，virtual synchrony，通过保证消息以一个所有参与者一致的视图顺序下进行处理来避免这个问题。

在一个现有的复杂系统的所有交互中引入序列号会产生很大的开销。因此，Chubby 提供了一种方式，使得只是在那些涉及到锁的交互中才需要引入序列号。锁的持有者可能在任意时刻去请求一个 sequencer，它是一系列用于描述锁获取后的状态的不透明字节串。包含了锁的名称，占有模式(互斥或共享)以及锁的世代编号。如果 client 期望某个操作可以通过锁进行保护，它就将该 sequencer 传送给 server(比如文件服务器)。接收端 server 需要检查该 sequencer 是否仍然合法及是否具有恰当的模式；如果不满足，它就拒绝该请求。{!那么到底是客户端是锁持有者还是服务器是持有者，保护的又是什么呢？client 端是锁的持有者，它能够随时去得到一个 sequencer(sequencer 实际上对锁状态的一种描述)，它希望这个锁可以保护它针对服务端进行的一个操作，因为可能有多个 client 去进行这个操作，这样呢为了避免上面提到的锁的乱序到达问题，因此必须提供一个机制避免这种情况，而 sequencer 就是为了解决这个问题，通过 sequencer 就可以知道一个锁是否依然有效，是否是一个过期的锁}sequencer 的有效性可以通过与 server 的 Chubby 缓存进行验证，如果 server 不想维护一个与 Chubby 的会话，也可以与它最近观察到的那个 sequencer 对比。sequencer 机制只需要给受影响的消息添加上一个附加的字符串，很容易解释给开发者。

尽管 sequencer 很容易使用，重要的协议也很少发生变化。但是仍存在一些不支持 sequencer 的 servers{!sequencer 机制应该是后来加入到 Chubby 系统中的，这样呢就可能存在一些老的 servers 不支持这种机制}，Chubby 为它们提供了一套不完美但是相对简单的机制{!即后面提到的 lock-delay}来降低延时及请求的 re-order 带来的风险。如果 client 以正常的方式释放了一把锁，正如期望的那样，对于其他客户端来说它就是立即可用的了。然而如果一把锁变为 free 状态是因为持有者 failed 或者不可访问了，那么锁服务器必须在一个被称为 lock-delay 的给定期限内禁止其他 client 获取它。client 可以设定一个任意上界的 lock-delay，

当前默认是一分钟；该限制可以避免一个故障的 client 无限期地占有一把锁。虽然并不完美，但是 lock-delay 策略还是使得那些未升级的 servers 和 clients 免受因日常的消息延迟和重启导致的影响。

2.5 事件

Chubby client 在它们创建句柄时可能订阅一系列的事件。这些事件通过来自 Chubby 库的后续调用异步地传输到客户端。事件包括：

- 文件内容改变—通常用于监控通过文件公布的某个服务的位置信息。
- 子节点的添加，删除或者修改—用于实现 Mirroring(2.12 节)。(除了允许发现新加入的节点，向客户端返回事件也使得监控临时性的文件而不影响它们的引用计数成为可能)
- Chubby master 故障恢复—提醒 client 某些事件可能已丢失，因此数据必须重新扫描。
- 句柄(或者是它的锁)失效—这通常意味着一个通信问题。
- 锁的获取—可以用来确定主本(primary)何时被选举出来。
- 来自于另一个客户端的冲突的锁请求—允许锁的缓存。

当相应的动作发生之后，事件才会被传递。因此如果一个客户端被通知文件内容已经改变，那么它之后去读取这个文件能够保证它一定能看到新的数据(或者是比事件发生时还要新的数据)。

最后两个事件很少被使用，现在看来，实际上应该去掉它们。比如对于 primary 选举，client 端通常需要与选出来的 primary 进行通信，而不是简单地知道 primary 存在与否就可以了；因此它们可以等待新的 primary 将地址写入文件所导致的一个文件内容改变事件即可。锁冲突事件理论上允许 clients 缓存其他 servers 持有的数据，通过 Chubby 锁来维护缓存一致性。当一个锁冲突事件发生时，实际就是告诉 client 结束使用与该锁相关的数据：它会结束那些等待中的操作，将修改刷新到原来的位置(home location){!实际上就是使用这种 Chubby 的锁冲突机制来实现缓存的一致性。应用场景就是这些客户端缓存来自服务端的数据，因此 home location 实际上就是数据在 server 端的存放位置}。目前为止，还没有人这样使用。

2.6 API

对于客户端来说，Chubble 句柄是一个支持各种操作的不透明结构。句柄只能通过 open()操作创建，通过 close()操作关闭。

Open()通过打开一个命名文件或目录来创建一个类似于 Unix 文件描述符的句柄。只有这个操作会使用节点名称，其他直接在句柄上进行操作。

节点名称通过相对于一个现有的目录句柄计算出来；库提供了一个一直有效的在"/"上的句柄。目录句柄避免了，因使用一个程序内部级别的当前目录，在包含多种抽象层次的多线程编程中的困难。{!正如我们所知，在 linux 中用户只需要

使用文件名即可 open 一个文件，这就隐含了一个当前路径的概念，但是在 Chubby 中没有这种概念，名称都是相对于同一个根目录路径，库默认使用”/”。}

在调用 open 时，client 可以用多种选项：

- 句柄如何使用(读，写以及锁；修改 ACL)；只有当客户端具有相对应的所需权限时句柄才会创建出来
- 需要传递的事件(参见 2.5)
- Lock-delay(参见 2.4)
- 应该(或者必须)创建一个新的文件还是目录。如果创建的是文件，调用者必须提供初始内容及初始 ACL 列表。返回值表示文件是否创建成功。

Close()操作关闭打开的句柄。这样关于该句柄的后续使用就是不允许的了。该调用永远不会失败。一个与之相关的调用 Poison()，能够不关闭句柄而使得后续的调用失败；这就允许 client 取消由某些线程 Chubby 调用，而不用担心会释放它们所访问的那些内存。

在一个句柄上的主要函数调用如下：
GetContentsAndStat()返回文件的内容和元数据。文件内容的读取是原子性的，同时必须整个读取。我们没有避免了文件的部分读取和写入以尽量防止生成大文件。一个相关的调用 GetStat()仅仅返回元数据，而 ReadDir()则会返回一个目录的子节点的名称及元数据。

SetContents()修改文件内容。可选择地，用户可以提供一个内容世代编号作为参数，以允许客户端模拟在文件上的 compare-and-swap 操作；只有当该编号等于当前值时内容才改变。文件内容的写入也是原子性的，同时也是整个写入地。一个相关的调用 SetACL()可以针对节点上的 ACL 列表执行与之类似的操作。

Delete()删除没有子节点的节点。

Acquire(),TryAcquire(),Release()用于申请和释放锁。

GetSequencer()返回一个用于描述该句柄持有的锁的状态的 sequencer。

SetSequencer()将一个 sequencer 与一个句柄关联。如果 sequencer 已经失效，那么在该句柄上的后续操作将会失败。

CheckSequencer()检查一个 sequencer 是否有效(见 2.4 节)。

如果在句柄创建之后节点被删除了，那么在句柄上的调用会失败，即使是该节点在后面又被重新创建出来。也就是说，句柄是与文件实例相关联的，而不是与文件名称。Chubby 可以将访问控制应用到所有的调用上，但是只是再 open 调用中进行检查(见 2.3 节)。

上面的所有调用为满足调用本身的其他需要都可以使用一个 operation 参数。该

参数可以用来持有数据以及与每次调用相关联的控制信息。尤其是通过该参数，client 可以：

- 支持使得调用可以异步化的回调函数
- 等待调用的结束，同时/或者
- 获得额外的错误及诊断信息

Client 可以使用这些 API 按照如下方式进行主本(primary)选举：所有的潜在主本打开锁文件，并尝试获取锁。最后只有一个会成功变成 primary，其他的就作为副本{!当多个请求成为 primary 时，如何保证最终只有一个会成功呢？因为消息的延迟，某些请求可能会滞后。如何解决这种问题？}。该 primary 将它的标识信息通过 SetContents()写入到锁文件，这样客户端和其他副本就能通过 GetContentsAndStat()获取到该信息了，可能是在一个文件改变事件(2.5 节)的响应函数中。理想情况下，该 primary 通过 GetSequencer()获取一个 sequencer，然后将它传给它所通信的 servers；这些 servers 通过 CheckSequencer()来验证它是否仍然是 primary。对于那些不能检查 sequencer 的 servers 则可能需要使用 lock-delay(见 2.4 节)。

2.7 缓存

为了减少读的流量，Chubby 客户端会在一个一致性的，写直达的内存缓存中缓存文件数据及节点元数据(包括文件缺失信息(file absence))。该缓存通过使用如下的租约机制来维护，通过 master 的失效通知来保证一致性，master 会维护一个用于描述客户端缓存内容的列表。该协议保证客户端要么看到 Chubby 状态的一个一致性视图要么是一个错误。

当文件数据或者元数据需要改变时，修改将会被阻塞到直到 master 已经向缓存了该数据的所有客户端发送了失效通知之后；该机制建立在下一节所描述的 KeepAlive RPC 基础上。收到一个失效通知后，客户端会刷新失效状态并在下一次 KeepAlive 调用中通知 master。当 master 已经确认每个客户端的缓存都已失效后，才会执行该修改操作，要么是因为客户端成功的返回了失效响应，要么是等待客户端缓存租约过期。

只需要一轮的失效通知，因为 master 会将那些对缓存失效状态还无法确认的 node(这个是指 Chubby 里的 node，而不是网络节点)看做是不可缓存的。这种策略允许读操作总是无延时的得到处理，这点非常有用，因为读操作要远远多于写。另外一种选择是：在失效通知期间阻塞所有访问该 node 的调用，这可以避免在失效通知期间过度急切的客户端对 master 造成轰炸式的访问，但是代价是增加了延迟。如果这会成为问题，可以尝试采用一种混合模式，在检测到过载时切换处理方式。

缓存协议很简单：在变更发生时首先使缓存数据无效，同时永不再更新它。更新它而不是使它失效可能一样简单，但是单纯的更新协议可能令人意外的低效；访问文件的客户端可能无限制地收到更新，这可能导致无数的不必要的更新{!因为没有人去访问它，因此这种更新实际上就是不必要的}。

尽管提供严格的一致性带来了额外的开销，但我们依然没有采用弱一致性，因为我们感到程序员会觉得它们更难使用。类似的，诸如虚同步(virtual synchrony)这样需要客户端在所有的消息中交换 sequence number 的机制， 在一个已经具有各种不同的现存协议的环境中也是不适合的。

除了缓存数据和元数据，Chubby 客户端还会缓存打开的句柄。因此，如果一个客户端打开了一个它之前已经打开的文件，只有第一次的 open()调用会引起一个到 master 的 RPC。缓存机制在某些方面进行了一些限制以保证不会影响客户端锁观察到的语义：临时文件的句柄在应用程序关闭它们之后就不能保持 open；那些允许锁定的句柄可以被重用，但是不能被多个应用程序句柄并发使用。最后一个限制存在的原因是，因为客户端为取消发送给 master 的 Acquire()调用可能使用 Close()或者 Poison()操作。

Chubby 的协议允许客户端缓存锁—这样锁的持有时间会比客户端所必需的期望持有时间要长。如果另一个客户端请求了一个冲突的锁，会产生一个事件通知给锁的持有者，这就允许持有者可以在需要的时候释放锁。

2.8 会话与 KeepAlives

一个 Chubby 会话在 Chubby cell 和 Chubby 客户端之间的一种关系，它存在于某某个时间间隔内，通过周期性的称为 KeepAlives 的握手维护。除非 Chubby 客户端通知 master，否则只要会话依然有效，那么客户端的句柄，锁及缓存数据就都是有效的。(然而会话维护协议为维护会话可能要求客户端回复一个缓存失效通知，如下)

在第一次与 Chubby 单元的 master 联系时，客户端会请求一个新的会话。在它正常结束或者会话空闲(在一分钟内没有打开的句柄及调用)它会显式地结束该会话。

每个会话都有一个与之相关的租约—一个延伸向未来的时间间隔，在这个期间内 master 保证不单方面的结束会话。该期间的结束阶段被称为会话租约过期。Master 可以自由的将过期时间向未来延迟，但是它不能将它在时间上前移。

Master 有如下三种时机进行续约：会话创建时，master 发生故障恢复时，当他响应来自客户端的 KeepAlive RPC 调用时。在收到一个 KeepAlive RPC 时，master 通常会阻塞该 RPC(不允许它返回)到该 client 的前一个租约接近过期。然后 master 允许该 RPC 返回给客户端，同时告知客户端新的租约过期时间。Master 可以以任意的延长过期时间。默认的延长时间是 12s，但是一个过载的 master 可能使用一个更高的值来减少它所需要处理的 KeepAlives RPC 调用。收到上一个 KeepAlives 调用的回复后，客户端会初始化一个新的 KeepAlives 调用。因此客户端可以保证通常只有一个 KeepAlives 调用阻塞在 master 端。

除了扩展客户端的租约外，KeepAlives 调用的回复还会被用于传递事件及缓存失效通知给客户端。当有一个事件或者缓存失效通知需要传递时，master 允许该

KeepAlives 调用尽早返回。在 KeepAlives 的回复上捎带事件信息，保证了客户端不响应缓存过期通知就不能维护一个会话，这也使得所有的 RPC 调用都是从客户端流向 master。这简化了客户端设计，同时也允许协议运行可以穿越那些只允许单向发起连接的防火墙。

客户端维护了一个本地租约过期时间，它要比服务端眼中的租约过期时间相对保守{!理解这一块具体可以参考论文<<Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency>>}。之所以不同于服务端的租约过期时间，是因为客户端必须在两方面做保守的假设：一是 KeepAlive 响应消息的传输时间，一是 master 时钟的超前度；为了维护一致性，我们要求 master 比 client 的时钟超前度必须在一定的常数因子之下。

如果客户端的本地缓存过期了，此时它就无法确定 master 是否已经结束了该会话。客户端就需要清空并禁用它的缓存，此时我们认为该会话处于危险(jeopardy)状态。客户端会继续等待一个称为宽限期(grace period)的时间区间，默认是 45 秒。如果在 grace period 结束之前，客户端和 master 又完成了一次成功的 KeepAlive 交互，那么客户端就会再次使它的缓存有效。否则，客户端就假设会话已过期。这样做就使得当 Chubby 单元不可访问时，Chubby 的 API 调用不会无限期的阻塞；在 grace period 结束，通信重新建立之前，这些调用会返回一个错误。

当 grace period 开始时，Chubby 库可以通过一个 jeopardy 事件通知应用程序。当已经获知会话发生了通信问题是，会产生一个 safe 事件来通知客户端去处理。这些信息使得应用程序在无法确认会话状态时可以保持静默，而且如果这个问题只是一个瞬时性的，那么客户端不需要重启就可以恢复。这对于很多启动开销很大服务避免服务的中断是很重要的。

如果客户端持有一个在某个节点上的句柄 H，同时因为关联会话的过期导致任何在 H 上的操作都失败了，那么所有的在 H 上的后续操作(除 Close()和 Poison()之外)也都会以同样的方式失败。客户端可以通过这一点来保证网络和 servers 的中断之后导致一系列的后续操作的丢失，而非任意可能的子序列，因此这就允许在某些复杂的操作的最后将其标记为 committed 状态。

2.9 故障恢复

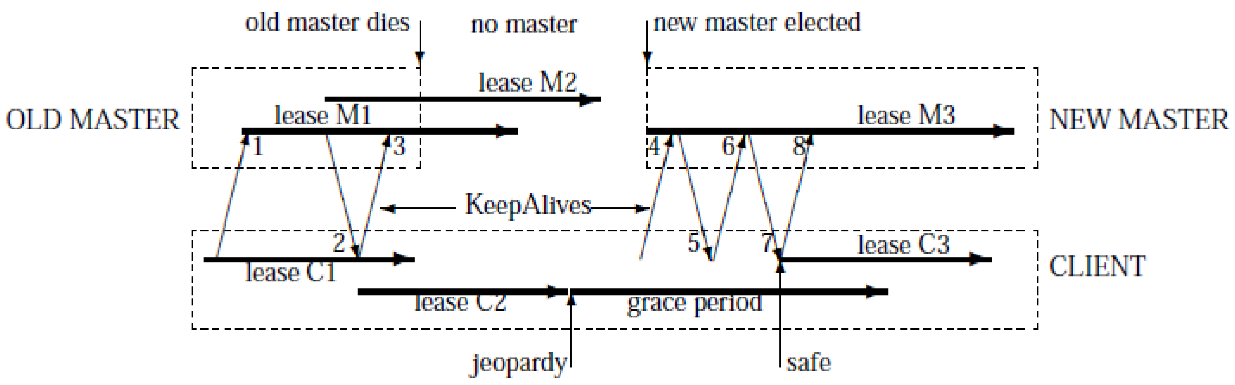


Figure 2: The role of the grace period in master fail-over

当一个 master 失败了或者失去了 master 身份时，它会丢掉它的关于会话，句柄及锁的所有内存状态。权威的会话租约计时器开始在 master 上运行{!如果 master 所在的机器挂了，如何运行它呢？实际上着可以看做是一种虚拟的租约计时器，因为 master 挂了，因此它不可能在接受任何修改操作，这样呢之前发出去的租约就可以认为是仍然有效的，因此这些租约是可以扩展到直到另一个 master 恢复的。虽然旧的 master 死了，新的 master 还未选出来，但是逻辑上我们可以认为有一个 master 存在，只是它无法支持对它的状态进行修改操作。}，直到一个新的 master 出来，这个会话租约计时器才会停止；这是合法的，因为这种方式等价于扩展客户端租约。如果一个 master 选举很快完成，那么客户端就可以在他们的本地(近似的)租约计时器过期之前联系新的 master。如果选举花了很长时间，在尝试寻找新 master 的同时客户端会刷新它们的缓存及等待进入 grace period。因此 grace period 使得会话可以超越正常的租约过期时间而能够在故障恢复期间仍能得以维护。

图 2 展示了一个漫长的 master 故障恢复中的一系列事件序列，在这个过程中客户端必须通过宽限期来保留它的会话。时间从左到右依次递增，但是时间没有按照比例画出{!即上图中各个阶段左右距离的跨度并不代表真实的时间长度}。客户端的会话租约用粗箭头表示，既有新老 master 眼中的租期 (上方的 M1-3)也有客户端眼中的租期(下方的 C1-3)。向上倾斜的箭头代表 KeepAlive 请求，向下倾斜的箭头代表针对它们的响应。原始的 master 具有客户端的会话租约 M1，与此同时客户端具有一个保守的估计 C1。在通过 KeepAlive 应答 2 通知客户端之前，master 允诺了一个新的租期 M2；之后客户端将它的租期延至 C2。在响应下一个 KeepAlive 请求之前，master 挂了，在另一个 master 选举出之前中间需经历一段时间。最后，客户端的租期 C2 到期了。之后，客户端刷新缓存，开始启动一个针对 grace period 的计时器。

在此期间，客户端无法确认它的租约是否已经在 master 端过期。它并不关闭这个会话，而是阻塞所有的应用程序调用以防止它们看到不一致的数据。在 grace period 开始时，Chubby 库向应用程序发送一个 jeopardy 事件，以使得应用程序

在能够确认会话状态之前保持静默。

最终一个新的 master 成功的选举出来。一开始该 master 使用一个对于它的前任对客户端的租约期限的保守估计 M3。新 master 收到的第一个来自客户端的 KeepAlive 请求(4)会被拒绝，因为它具有错误的 master epoch 编号(细节稍后描述)。重试请求(6)会成功，但是通常不会扩展 master 的租约期限，因为 M3 是一个保守值{!为何是保守值就不去扩展它呢？}。然而响应(7)允许客户端再一次扩展它的租约(C3)，同时可以选择通知应用程序它的会话已经不再处于危险期(jeopardy)。因为 grace period 长的足以跨越 C2 的结束及 C3 的开始这段时间，对于客户端除延迟之外其他的都是透明的。假若 grace period 小于这个时间段，那么客户端会直接丢弃该会话并向应用程序报告错误。

一旦一个客户端联系上新的 master，客户端库就会和 master 相互协作提供给应用程序没有故障发生的假象。为了实现这个，新的 master 就必须重建它的前任 master 内存状态的一个保守近似。一部分通过读取保存在硬盘上的数据(会通过普通的数据库备份协议来进行备份)来完成，一部分通过从客户端获取状态，一部分再通过保守的估计来完成。数据库会记录每个会话，持有的锁及临时文件。

新的选举出的 master 执行如下步骤：

1. 它首先选择一个新的 epoch number，在每次调用中客户端都需要出示该编号。Master 会拒绝那些使用老的 epoch number 的客户端调用，同时会提供出新的编号。这就保证了一个新的 master 不会响应原本发送给旧 master 的非常老的包，即使是新老 master 运行在同一台机器上。
2. 新的 master 响应 master 定位请求，但是最开始它并不处理收到的会话相关操作。
3. 为那些记录在内存和数据库中的会话和锁建立内存数据结构。会话租期被延至前一个 master 所曾经使用的最大值。
4. Master 开始让客户端执行 KeepAlive，但是仍不允许其他的会话相关操作。
5. 为每个会话产生一个故障恢复事件，这会导致客户端刷新它们的缓存(因为它们可能曾经丢失了一些缓存失效通知)，同时警告客户端某些事件可能已经丢失。
6. Master 等待每个会话已经对这个故障恢复事件做出响应，或者它的会话过期。
7. Master 开始允许各种操作的处理。
8. 如果客户端使用一个先于故障恢复点创建的句柄(通过句柄中的某个编号值可以判断出来)，master 会在内存中创建该句柄，并执行该调用。如果这样创建出的句柄被关闭了，那么 master 会在内存中记录下来，保证它不会在这个 master 存活期间被再次创建；这就保证了一个延迟或者重复的网络包不能意外地创建出一个已经关闭的句柄。一个失败的客户端可以在未来的世代中重新创建一个已经关闭的句柄，但是因为该客户端已经出错失败了因此这样是没有问题。
9. 一段时间之后(比如一分钟)，master 删除那些没有打开的句柄的临时文件。在故障恢复之后的这个时间段内，客户端应该刷新它们在临时节点上的句柄。这种机制有一个问题，如果临时文件的最后一个客户端在故障恢复期间丢失了会话的话，那么它们可能不会及时的消失。

可能令读者吃惊的是，故障恢复代码虽然远比系统其他部分代码的执行机会要少，但是却是很多有趣 bug 的丰富来源。

2.10 数据库实现

第一版的 Chubby 使用带复制的 Berkeley DB 作为它的数据库。Berkeley DB 提供一个用于将字节串映射到任意字节串的 B 树。我们提供一个 key 的比较函数，该函数会首先根据路径名称里的单元数进行排序；这就允许节点以它们的路径名称作为 key，同时保证了排序后兄弟节点是相邻的。因为 Chubby 没有使用基于路径的权限，每个文件访问只需要在数据库中进行一次查找即可。

Berkeley DB 使用一个分布式一致性协议来将它的数据库日志复制(replicate)到一个服务器集合上。只要在加上 master 租约，就跟 Chubby 的设计匹配了，这就使得我们的实现变得很直接。

虽然 Berkeley DB 的 B 树相关的代码已经被广泛使用而且很成熟了，但是复制(replication)相关的代码却是最近才加入的，而且很少有用户。软件维护者肯定优先维护改进那些最流行的产品特性。在 Berkeley DB 的维护者解决我们遇到的问题期间，我们感到使用其与复制(replication)相关的代码所承担的风险超过了我们的预期。最终，我们使用日志预写(write ahead log)和快照(snapshotting)实现了一个类似于 Birrell 的简单数据库。与之前相同，数据库日志通过一个分布式一致性协议在不同副本间分发。Chubby 只使用了 Berkeley DB 很少的一些 feature，因此这个改写大大简化了我们的系统；比如我们可能需要原子操作，但是我们并不需要通用的事务。

2.11 备份

每隔几个小时，每个 Chubby 单元的 master 就将它的数据库的一个快照写入到另一栋楼里的 GFS 文件服务器。让它们处在不同的楼里可以保证该备份可以在楼宇毁坏时仍得以保存，同时也避免了在系统中引入循环依赖；在同一栋楼里的 GFS 单元潜在的依赖于 Chubby 单元选择它的 master。

备份既提供了灾难恢复，又提供了一种方式初始化那些新的被替换的副本而不会对服务中的那些副本带来负载。

2.12 Mirroring

Chubby 允许将一组文件集合从一个单元镜像到另一个单元。Mirroring 速度很快，因为文件很小而且事件机制会在文件加入，删除或修改时立即通知镜像处理代码。在不存在网络问题的情况下，变更可以在 1 秒内反映到世界范围内的多个镜像中。如果某个镜像是不可到达的，它会保持不变直到连接恢复。通过比较校验和就可

以识别出发生更新的文件。

镜像机制通常被用来向分布在世界各地的计算集群拷贝配置文件。一个特殊的名为 global 的 Chubby 单元，包含一个子树/ls/global/master，该子树会被镜像到所有其他的 Chubby 单元的子树/ls/cell/slave 中。该名为 global 的 Chubby 单元非常特殊，因为它的 5 个副本是分散在世界上相聚遥远的地方，因此大部分的国家和地区几乎都可以访问它。

在从名为 global 的 Chubby 单元镜像的那些文件，有 Chubby 自己的访问控制列表，有 Chubby 单元和其他系统用于向监控服务广播它们的存在的文件，有允许客户端定位大数据集合比如 Bigtable 单元的指针，以及很多其他系统的配置文件。

3. 扩展性机制

Chubby 的客户端是独立的进程，因此 Chubby 必须处理比人们想象的还要多的客户端。我们曾观察到 90000 个客户端直接与 Chubby master 通信—这远远大于相关的机器数。因为一个 Chubby 单元只有一个 master，而且它的机器配置与这些客户端一样，于是客户端能以巨大的优势将 master 压垮。因此，最有效的扩展性技术是最大程度地减少客户端与 master 的通信。假设 master 没有严重的性能问题，在 master 请求处理上的细微改进几乎不起作用。我们使用如下几种策略：

- 我们可以创建任意数量的 Chubby 单元，客户端总是使用一个就近的单元(通过 DNS 找到它)避免它使用一个远端的 Chubby 单元。我们的典型部署中，一个 Chubby 单元通常为一个数据中心的数千台机器服务。
- 在 Master 处于重负载的情况下，它可能将租约有效期从默认的 12 秒增加到 60 秒左右，这样就减少了它需要处理的 KeepAlive PRC 调用。(目前为止 KeepAlive 一直是最主要的请求类型，见 4.1 节，并且未能及时处理它们是一个超负载的服务器典型的失败模式；客户端对其他调用中的延迟变化相当不敏感)
- Chubby 客户端缓存文件数据，元数据，文件缺失信息及打开的句柄以降低它们在 master 上执行的调用。
- 我们使用协议转换服务器将 Chubby 协议转换为更简单的其他协议比如 DNS。后面我们还会讨论这点。

下面我们描述两种很熟悉的机制代理和 Partitioning，通过它们我们希望可以让 Chubby 更具扩展性。目前我们还未在产品系统中使用它们，但是它们已经设计出来，后面可能很快就会投入使用。我们没有必要考虑超过 5 倍的可扩展性：首先，能够放入一个数据中心的机器数目有限，依赖于单个服务实例的机器数也有限。其次，因为我们为 Chubby 的客户端和服务端使用了类似的机器，硬件升级能够增加单个机器上的客户端数目，同时也会增加当个服务端的处理能力。

3.1 代理

Chubby 协议可以由可信进程作为代理(在两边使用相同的协议)，将请求从客户端

传到 Chubby 单元。一个代理通过处理 KeepAlive 和读请求来降低服务端负载；它无法降低传过代理缓存的写操作流量。即使具有很强悍的客户端缓存，写操作流量也远小于 Chubby 正常工作负载的 1%(见 4.1 节){!客户端缓存会降低读操作的流量，即使在这种情况下写操作也占不到 1%，这就说明写操作不是瓶颈，因此没必要为它而优化}。因此代理可以大大的增加客户端的数目。如果一个代理处理了 Nproxy 个客户端请求，KeepAlive 流量会降低 Nproxy 倍，而 Nproxy 的值可能上万甚至更大。代理缓存可以降低最多是 read-sharing 平均大小的读流量—大概是 1/10(见 4.1 节)。但是因为读流量目前只占了 Chubby 负载的 10%不到，因此远不如 KeepAlive 流量节省的影响大。

代理增加了一个额外的 RPC 用于写操作及第一次的读操作。人们可能觉得这使得 Chubby 单元的可用性与之前相比至少降低了 2 倍，因为每个被代理的客户端现在依赖于两台机器：它的代理和 Chubby master。

读者可能注意到 2.9 节的故障恢复策略在代理机制下并不理想。我们会在 4.4 节描述这个问题。

3.2 Partitioning

正如 2.3 节提到的，Chubby 的接口选择保证了一个 Chubby 单元的名字空间可以划分到不同的服务器。尽管目前我们还不需要这样做，但是目前的代码已经支持根据目录划分名字空间。如果开启划分，一个 Chubby 单元可能会由 N 个分区组成，每个分区具有一组副本和一个 master。在目录 D 下的每个节点 D/C 将会被存储到分区 P(D/C)=hash(D)mod N。需要注意的是 D 的元数据可能会存储在一个不同的分区 P(D)=hash(D’) mod N，此处 D’代表 D 的父亲。

分区用来启用那些分区之间通信很少的大的 Chubby 单元。尽管 Chubby 没有硬链接，目录修改时间及跨目录重命名操作，有些操作仍然需要分区间的通信：

- ACLs 本身是文件，因此一个分区可能使用另一个分区做权限检查。但是 ACL 文件被缓存后，只要 Open()和 Delete()操作需要进行 ACLs 检查；而且大多数客户端只是读取那些不需要 ACL 的公共访问文件。
- 当目录被删除时，一个跨区的调用可能需要保证目录是空的

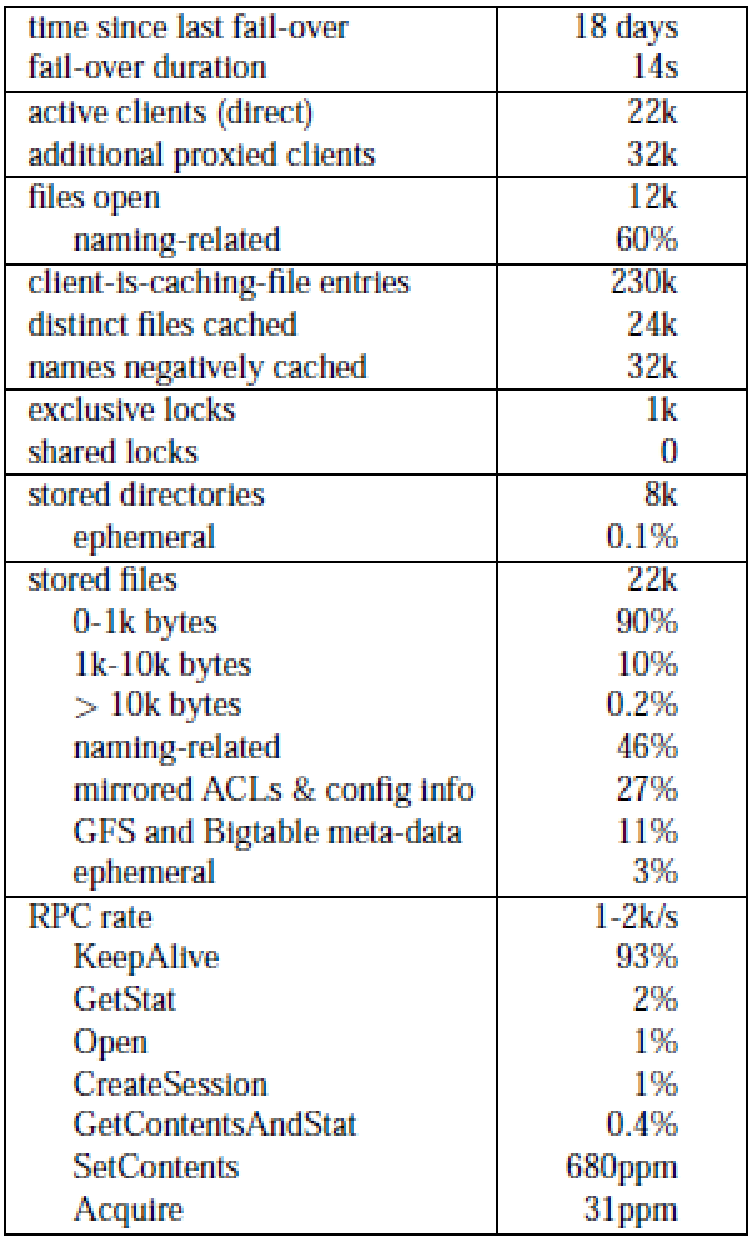
因为每个分区独立处理大部分的调用，我们预期这种通信不会对性能和可用性造成大的影响。

除非分区数目 N 很大，我们认为每个客户端将会与大部分分区进行联系。因此，分区将任意一个分区上的读写量降低到原来的 1/N，但是并不会降低 KeepAlive 的流量。如果 Chubby 需要处理更多的客户端，我们的策略将是联合使用代理和分区。

4. 实际应用、意外(surprises)和设计错误

4.1 应用与行为统计(use and behaviour)

下面的表格给出了某个 Chubby 单元在某一时刻的统计信息；其中 RPC 的频率是通过一个 10 分钟的区间计算出来的。下面的这些数字对 google 的很多 Chubby 单元来说都很典型：



time since last fail-over	18 days
fail-over duration	14s
active clients (direct)	22k
additional proxied clients	32k
files open	12k
naming-related	60%
client-is-caching-file entries	230k
distinct files cached	24k
names negatively cached	32k
exclusive locks	1k
shared locks	0
stored directories	8k
ephemeral	0.1%
stored files	22k
0-1k bytes	90%
1k-10k bytes	10%
> 10k bytes	0.2%
naming-related	46%
mirrored ACLs & config info	27%
GFS and Bigtable meta-data	11%
ephemeral	3%
RPC rate	1-2k/s
KeepAlive	93%
GetStat	2%
Open	1%
CreateSession	1%
GetContentsAndStat	0.4%
SetContents	680ppm
Acquire	31ppm

从表中可以看到如下几点：

- 很多文件被用于命名服务；见 4.3 节
- 配置，访问控制，及元数据文件(类似于文件系统的超级块)很普遍
- Negative caching 很明显
- 230k/24=10，平均看来大概每个缓存文件有 10 个客户端使用
- 很少有客户端持有锁，共享锁也很少见；这与锁主要用于 primary 选举及在

副本间划分数据是一致的

- RPC 流量主要由会话 KeepAlive 组成；之后是一些写操作(缓存未命中的)；很少由写操作或锁的申请。

现在可以简短描述下 Chubby 单元失效的典型原因。假设如果我们乐观的认为只要 master 还在提供服务就认为 Chubby 单元就是处于可工作状态，那么在我们的 Chubby 单元中，在几周的时间内总共记下了 61 次失效，总共大概是 700 个 Cell-day 的时间{!Cell-day 实际上是类似于人月(人月神话中的人月)的概念，因为观察的是多个 Cell 的出错的总的情况，比如可能观察的是 5 个 Chubby 单元在 10 天内的情况，那么换算成 Cell-day，就是 50 个 Cell-day}。排除掉因数据中心停机维护引起的失效外，其他的失效原因如下：网络拥塞，维护，超载，操作错误，软件和硬件问题。大部分的失效持续 15 秒或者更少的时间，其中有 52 个低于 30s；我们大部分的应用程序不会因为 Chubby 单元 30s 的失效而受到影响。剩余的 9 次失效，其中 4 个因为网络维护引起，2 个怀疑是因为网络拥塞，2 个是因为软件错误，还有 1 个是因为负载过重引起。

在很多 cell-year 的运行中，发生过 6 次数据丢失，其中 4 次是因为数据库软件错误，2 次是因为操作失误；没有与硬件失败相关的。更讽刺的是，因升级引起操作错误却是为了避免软件的错误。我们有 2 次纠正过因非 master 副本引起的数据损坏。

Chubby 的数据都可以放入内存中，因此大部分的操作代价都很小。我们的生产服务器的平均请求延迟通常不到 1ms，在 Chubby 单元负载过重时，延时会显著增加，很多会话会被丢弃。负载过重同时是因为同时有很多(>90000)会话处于活动状态，但也可能是由异常情况引起的：比如大量客户端并发产生数百万的读请求(如 4.3 节所述)，或者是因为客户端库的问题导致部分写操作的缓存无效而引起每秒上万次的请求。因为大部分的 PRC 调用都是 KeepAlive，通过增加会话租约期限(第 3 节)服务端可以将多个活动客户端的平均请求延迟维护在较低的一个水平上。

客户端的 RPC 读延迟受限于 RPC 系统和网络；在一个本地 Chbby 单元中，它的值可能小于 1ms，但是 Chubby 单元距离客户端极遥远时，该值可能就是 250ms。写操作(包括锁操作)因为要更新数据库日志因此可能还要慢 5-10 毫秒，但是如果有一个最近发生故障的客户端缓存了该文件，这个时间可能要上升至数 10 秒{!要等待它的缓存租约过期}。但是这种写操作上的变化对于平均请求延迟具有很少的影响，因为写操作并不经常发生。

只要会话未被丢弃，客户端很少受延迟变化的影响。需要注意的一点，为制止恶意的客户端，我们在 open()操作中增加了人为的延迟(见 4.5 节)；只有当延迟超过 10 秒并且反复出现时，开发者才会注意到它们。我们发现提高 Chubby 扩展性的关键不在于服务端性能；降低与服务端所需的同学会有更好的效果。我们没有为读写服务器的代码进行特殊的优化，只是检查一下不会有严重的 bug，然后注重于可扩展性机制上，这样会更有效。另一方面，开发者们确实发现过因本地的 Chubby cache 的一个性能 bug，而导致客户端每秒可能进行了成千上万次的读操作。

4.2 java 客户端

Google 的基础架构设施大部分都是用 C++实现的，但是用 Java 实现的的系统也在日益增长。这种趋势给 Chubby 带来了一些问题，而 Chubby 本身有一个复杂的客户端协议及一个非平凡的客户端库。

Java 带来了很好的移植性，但是也因为要把它与其他语言进行连接而带来了一些额外的花费。通常 Java 访问非原生(no-native)库的机制是 JNI，但是它通常被认为效率低下而繁琐。我们的 Java 程序员们并不喜欢使用 JNI，为了避免使用它，他们更喜欢将库翻译成 Java，并维护它们。

Chubby 的 C++客户端库大概 7000 行代码(与服务端差不多)，而且客户端协议很精细。维护这样的一个 Java 实现的库，需要小心谨慎及额外的成本，而一个没有缓冲的实现甚至会搞垮服务器。因此，我们的 Java 用户运行了多个协议转换服务器，提供一个类似于 Chubby 客户端 API 的简单 RPC 协议。即使事后看来，如何避免去编写，运行并维护一个额外的服务器仍然不是那么明显的。

4.3 作为 name service 使用

尽管 Chubby 是为锁服务而设计的，我们发现它最流行的应用却是作为名字服务器。

在 DNS(普通 Internet 名字解析系统)中，缓存机制是基于时间的。DNS 记录有一个有效期(TTL：time-to-live)，如果在这个时间段内 DNS 数据没有被刷新，它们就会被丢弃。通常很容易选择一个合适的 TTL 值，但是如果需要能快速替换失败的服务，此时需要设置较小的 TTL，但 TTL 太小的话，有可能会使得 DNS 服务器超载。

比如，开发者运行一个具有数千个进程的作业(job)是很常见的，如果这些进程相互之间都需要通信，这将会导致一个平方级别的 DNS 查找。我们可能想使用一个 60s 的 TTL，这既可以让那些异常的客户端能够在可以接受的延迟内被替换掉，同时在我们的环境里也不认为该时间过短。在这种情况下，为了维护一个具有 3000 个客户端的作业，可能需要每秒高达 150,000 次的查找(lookup){!总共 3000*3000，除以 60 就是 150,000}。(作为对比，一个双核 2.6G 的 Xeon DNS 服务器每秒可以处理大概 50,000 个请求)。Job 规模越大，问题就会越严重，同时还可能有多个 job 同时在跑。在引入 Chubby 之前，这种 DNS 的负载波动对于 Google 来说一直是一个严重的问题。

与之相比，Chubby 的缓冲使用显式的失效，这样在没有变化发生时，一个固定的会话 KeepAlive 请求就能维护客户端任意数目的缓冲记录。我们曾经观察到一个双核 2.6G 的 Xeon Chubby master 处理直接与它通信的 90,000 个客户端(没有代理的情况下)；这些客户端包括在具有上文提到的通信模式的 Job 中。这种不用轮询每个名字就可以快速进行名字更新的能力{!何意？如何实现，DNS 又是如何

进行名字更新呢？对于 DNS 来说，名字信息更新后，真正反映到客户端，是需要客户端进行查询之后，而这个过程可能需要轮询多级 DNS 服务器，之后这些 DNS 服务器才会从权威的 DNS 服务器拥有该记录，而对于 Chubby 来说可以直接访问保存这些信息的那个文件即可}，是如此吸引人，现在公司的大部分系统都使用由 Chubby 提供的名字服务。

尽管 Chubby 的缓存机制允许单个单元为大量的客户端提供服务，当时峰值负载仍可能是一个问题。在我们最初部署基于 Chubby 的名字服务时，启动一个 3000 个进程的 job 时(因此会产生 9000,000 个请求)可能会将 Chubby master 压垮。为解决这个问题，我们将名字条目分组进行批量处理，这样一次查找可能返回并缓存 job 里相关进程的大量(通常是 100 个)的名字映射。

由 Chubby 提供比一个普通名字服务更精确的缓存语义；名字解析只需要定期的通知而不需要完全一致。因此，这就可以通过特别为名字服务引入一个简单的协议转换服务器来降低 Chubby 的负载。如果我们之前能预见到将 Chubby 作为名字服务器使用，为了避免这样一个简单但是没有必要的额外的服务器，那么可能就会更快地去实现一个完整的代理。

现在有一种更进一步的协议转换服务器：Chubby DNS 服务器。它使得存储在 Chubby 内的名字数据可以被 DNS 客户端访问。这种服务器在简化从 DNS 名称到 Chubby 名称的转换，以及兼容现有的应用程序(比如浏览器)这两个方面都很重要。

4.4 故障恢复的问题

master 故障恢复的原始设计(2.9 节)中需要 master 将新的会话在创建时写入数据库。在 Berkeley DB 版的锁服务器中，在很多进程同时启动时，这种会话创建带来的开销就成了一个问题。为了避免超载，可以对服务器进行修改，不再是在会话第一次创建时将它存入数据库，而是在它首次试图进行修改、获取锁或者打开一个临时文件时。另外，活动的会话在每个 KeepAlive 上以一定的概率存入数据库。这样对于只读性会话的写入就能均摊到一个时间段上。

尽管对于避免超载这是必要的，但是这种优化也带来了一些问题，那些年轻的只读会话可能没有被记录数据库中，因此在故障恢复发生时就可能会被丢失。尽管这些会话没有持有锁，但仍然是不安全的；如果所有已记录的会话在被丢弃的会话过期前被新的 master 接受到(check in)，那些被丢弃的会话可能会在一段时间内读到脏数据。实际中这很少发生；但是在一个大的系统中，几乎可以肯定总是会有会话 check in 失败，不管怎么样这都迫使 master 要等待最大租约过期。因此，为了避免这种影响，以及在当前机制引入代理后产生的复杂性，我们修改了故障恢复的设计。

在新的设计中，我们完全避免了在数据库中记录会话，取而代之的是在 master 重建句柄时(2.9 节 8)以同样的方式对会话进行重建。一个新的 master 现在必须等待一个完整的最坏情况下的租约过期{!2.9 节中，master 可能不需要等待这样一个时间，只要它收到了所有会话的应答即可，但是新的设计因为没有保存会话，

因此也就不知道到底有哪些会话，所以别无选择}，才能允许操作继续处理，因为它无法知道是否所有的会话都已经 check in(2.9 节 6)。这在实际中影响也很小，因为可能并非所有的会话都需要 check in。

一旦会话可以在免磁盘状态的情况下就可以创建，代理服务器就可能会管理 master 不知道的会话。有一个只有代理可用的操作，即允许它们改变有锁所关联的会话。这就允许可以将客户端从一个失败的代理移到另一个。master 端唯一需要做的修改是，在新的代理有机会声明它们之前，保证不会放弃一个与代理会话相关的锁或临时文件句柄。

4.5 客户端的滥用(abusive client)

Google 的产品团队可以自由创建自己的 Chubby 单元，但是这样增加了维护负担及额外的硬件资源开销。因此很多服务使用共享的 Chubby 单元，这使得将正常的客户端与非正常客户端隔离变得很重要。Chubby 只是用于公司内部，因此针对它的恶意的拒绝服务式的攻击很少会发生。但是开发者的失误，理解上的偏差以及不同的预期都可能会导致类似恶意攻击的后果。

某些补救方法可能过于繁重。比如，我们 review 产品团队计划使用 Chubby 的方式，在 review 通过前，拒绝它们对共享的 Chubby 单元的访问。这种方式的一个问题是开发者通常不能预测它们的服务在未来的使用方式及增长速度。读者可能注意到这样的一个有点讽刺性的事实，我们本身就没有预测到 Chubby 将会如何被使用。

我们 review 的最重要的一个方面是要判断对 Chubby 任何资源(RPC 调用，磁盘空间，文件数)的使用是否是随项目所处理的用户数或者数据量的增加成线性增长(或者更糟糕的增长关系)。任何的线性增长率必须能通过一个补偿参数将 Chubby 的负载降低到合理的边界上。不过，我们早期的 review 并不彻底。

一个相关的问题是在大多数的软件说明文档中缺乏性能方面的建议。由一个团队所写的模块可能会在一年后被另一个团队重用，就可能产生灾难性的后果。有时很难向接口设计者解释，告诉他们必须要修改他们的接口，不是因为他们接口设计地很差，而是因为这可能让其他开发者很难意识到 RPC 的开销。

下面我们列出一些我们遇到的问题：
缺乏有效的缓存机制。起初我们并没有意识到缓存缺失的文件的重要性，也没有重用已打开的句柄。尽管在培训时已经说明，但是我们的开发者还是经常在写循环中当某个文件不存在时会进行无限制的重试，或者是通过重复的打开关闭一个文件对它进行轮询(实际上他只需要打开一次即可)。

起初，我们通过在应用程序在短时间内做多次 open()操作的尝试时，给它引入一个指数性增长的延迟，来解决这个重试问题。在某些情况下，这能暴露出开发者这种已知性的 bug，{!即在短时间内重复打开文件，根据设计是要避免的，但是开发者虽然知道这点但在实现中可能又忽略了这点。实际上通过修改设计，使得

开发者无需记忆这种原则，而是允许这种使用方式，是更廉价的方式。}但是这需要我们花更多是时间去培训。最后让重复性的 open()调用变得廉价(即通过缓存缺失文件或重用打开的句柄)会更简单些。

Quota 的缺乏。Chubby 并不是一个为大数据量而设计的一个存储系统，因此没有存储限额(Quota)。现在看来，这种想法太天真了。

Google 曾经有一个项目，该项目有一个负载追踪数据上传的模块，会在 Chubby 中存储某些元数据。起初上传行为很少发生，而且限制在某几个人中，因此所用的空间有限。但是，后来另外两个服务开始使用该模块作为监控大量用户的上传行为的一种方式。最终，随着他们服务的增长，Chubby 到达了极限：追踪一个用户行为就需要去写一个 1.5M 的文件，而由这项服务所用的空间也超过了所有其他服务使用空间的总和。

我们引入了一个对于文件大小的限制(256k 字节)，同时鼓励这项服务迁移到更合适的存储系统上去。但是很难去推动那些维护产品系统的繁忙的人们，去做一个很大的变更—为了把数据迁移到别处花了大概一年时间。

发布/订阅。曾经有几次尝试使用 Chubby 的时间机制作为一个类 Zephyr 的发布/订阅机制。Chubby 重量级的一致性保证以及为维护一致性使用的失效机制(而没有用更新机制)使得它在大多数使用场景中都很慢而且效率很低。幸运的是，所有这样类似的应用，都在重新设计应用程序的代价变得太大之前得以叫停。

4.6 吸取的经验教训

这里我们列出一些经验教训以及一些如果有机会可能会做的一些设计方面的变更：

开发者很少考虑可用性。我们发现我们的开发者很少考虑失败情况，同时他们倾向于将 Chubby 看做是一个总是处于可用状态的服务。比如，开发者曾经建立过一个使用了数百台机器的系统，在 Chubby 选举出新的 master 后就启动了一个数十分钟的恢复处理过程。这使得一个单点失败产生的影响，在机器数及时间上都扩大了数百倍。我们更希望开发者为短暂的 Chubby 失效做好计划，这样它们的应用就几乎不受此类事件的影响。这也是使用粗粒度锁的一个原因，在 2.1 节讨论过这个问题。

另外开发者对于服务在线(being up)以及服务可用(available)之间的区别缺乏认知。比如 global 的 Chubby 单元(见 2.12 节)几乎总是处于在线(up)状态，因为很少会出现相距遥远的两个数据中心同时 down 掉。然而，客户端所感受到的它的可用性往往要低于本地的 Chubby 单元的可用性。首先，本地 Chubby 单元很少与 client 隔离，第二，尽管本地 Chubby 单元可能因为机房维护而停机，但是这种维护也同样会影响到 client，这样此时 Chubby 单元的不可用对于 client 的不可见的。

我们在 API 上的选择也会影响到开发者对 Chubby 失效的处理方式。比如，Chubby

提供一个事件允许客户端检测到 master 故障恢复的发生。本来这是为了让客户端检查某些可能的变化，因为某些事件可能会被丢失。不幸的是，很多开发者会选择在收到此类事件时，停掉他们的应用程序，因此这大大降低了他们系统的可用性。如果只是发送一个”文件内容”事件或者是保证在故障恢复期间不丢失事件可能会更好一些。

目前我们使用了三种机制来防止开发者会 Chubby 可用性过分乐观，尤其是对 global 的 Chubby 单元。首先，像前面提到的(4.5 节)，我们会 review 产品团队计划如何使用 Chubby，并建议他们不要将他们产品的可用性与 Chubby 绑定地太紧。第二，我们现在提供一些执行某些高级任务的库，使得开发者可以自动地从 Chubby 失效中隔离。第三，我们利用每次 Chubby 失效的事后分析作为一种手段，不仅是消除 Chubby 及操作过程中的 bug，还要降低应用程序对 Chubby 可用性的敏感性—这都能帮助提高我们系统整体的可用性。

细粒度的锁是可以被忽略的。在 2.1 节的最后，我们提出了一个提供细粒度锁机制的方案。令人吃惊的是，直到目前为止，我们还没有需要去实现这样一个机制；开发者通常发现为了优化应用程序，他们必须移除不必要的通信，这通常意味着需要寻找一种粗粒度的锁机制。

糟糕的 API 选择可能产生无法预料的影响。我们大部分的 API 都演化的很好，但是有一个错误很突出。我们取消长时间运行的调用是 Close()和 Poison() 这两个 RPC 调用，它们同时也会丢弃句柄的服务端状态。这就使得能够获取锁的句柄不能被共享，比如被多个线程。我们可能要增加一个 Cancel() RPC 来允许对打开的句柄的更多的共享。

RPC 使用影响了传输协议。KeepAlive 既被用于刷新客户端的会话租约，也用于传递事件及缓存从 master 到客户端的失效通知。这个设计有一个影响：客户端在没有应答缓存失效通知的情况下不能刷新它的会话租约。

这看起来还算合理，除了在我们的协议选择中引入一些限制。TCP 的拥塞回退不关心更高层的超时(比如 Chubby 租约)，这样基于 TCP 的 KeepAlive 在网络拥塞时可能会导致大量会话的丢失。这样我们不得不通过 UDP 发送 KeepAlive RPC 而不是 TCP；UDP 本身没有拥塞避免机制，因此我们更希望只有当更高级的时间边界必须满足时才使用 UDP。

我们觉得可以为协议提供一个额外的基于 TCP 的 GetEvent() RPC，可以用它在正常情况下进行事件和失效通知，采用与 KeepAlive 相同的使用方式。KeepAlive 仍然会包含一个未回复的事件列表，因此事件最终必须得到回复。

5. 与相关工作的对比

Chubby 是基于一些早已成熟的想法之上。Chubby 的缓存设计基于在分布式文件系统上的工作[10]。会话和缓存 token 类似于 Echo 里的行为[17]；会话减少了 V

系统中的租约[9]开销。关于通用锁服务的想法可以在 VMS[23]里找到，尽管该系统起初使用了一个具有低延迟交互的专用高速互联网络。与它的缓存模型类似，Chubby 的 API 基于一个文件系统模型，包括这种比纯文件更方便的类文件系统名字空间的想法也是来自[8,21,22]。

在性能及存储能力上，Chubby 不同于诸如 Echo 或者 AFS 这样的分布式文件系统：客户端不会读、写及存储大量数据，也不期望高吞吐率，也不要求缓存数据的低延迟。但是它希望具有一致性，可用性及可靠性，在性能不是那么重要的情况下这些属性就比较容易达到。因为 Chubby 的数据库很小，因此我们能在线存储它的多个拷贝(通常有 5 个副本及一些备份)。每天我们会进行多次备份，并且每隔几个小时就会通过数据库状态的 checksum 在副本之间进行比较。通过对普通文件系统所需的性能及存储需求上的弱化，允许我们可以通过一个 Chubby master 为成千上万的客户端提供服务。通过提供一个很多客户端可以共享信息及协调工作的中央节点，我们解决了我们的系统开发者所面对的一类问题。

关于文件系统和锁服务器方面有大量的可参考文献，我们无法提供一个详尽的比较，因此我们只是选择了其中的一个 Boxwood 的锁服务器来进行比较，因为它是最近设计的，同时也旨在用于松耦合的环境中，而且它的很大设计都与 Chubby 不同，有些很有趣，有些并不是很重要。

Chubby 实现了锁，一个可靠的小文件存储系统，一个会话/租约机制。与之相比，Boxwood 将这些分成了三块：锁服务，Paxos 服务(可靠的状态存储机制)，一个独立的失败检测服务。Boxwood 系统使用到了这全部的三个模块，但是某些系统可能只需要其中的某个模块。我们认为这种设计上的不同源于目标用户的不同。Chubby 计划用于各种不同的用户及应用程序；它的用户可能从创建分布式系统的专家到那些写管理脚本的初学者。在我们的环境里，一个具有令人熟悉的 API 的大规模锁服务看起来更具吸引力。与之相比，在我们看来 Boxwood 主要提供了一个工具集，更适合于那些工作在需要共享代码但无需一块使用的项目上的少数高级开发者。

很多情况下，Chubby 都提供了一个比 Boxwood 更顶层的借口。比如，Chubby 合并了锁和文件名字空间，而 Boxwood 的锁名称就是简单的字节序列。Chubby 客户端默认缓存文件状态；Boxwood 的 Paxos 服务的一个客户端可以通过锁服务实现缓存机制，但是可以使用 Boxwood 本身提供的锁服务。

因为是面向不同的期望应用，这两个系统在默认参数有着限制的不同：每个 Boxwood 失败检测器每隔 200ms 会与客户端通信一次，超时时间是 1s；Chubby 的默认租约时间是 12s，KeepAlive 没 7s 进行一次。Boxwood 的子部件使用 2 个或者 3 个副本来实现可用性，而我们通常在每个单元中使用 5 个副本。然而，这些选择并没有暗示着设计方面的更深度的差别，而只是说明了系统如何为容纳更多的客户端，或者适应与其他项目的共享机柜的不确定性，而必须如何去调整这些参数。

一个更有趣的区别是 Chubby 引入的宽限期(grace period)，而 Boxwood 没有这个。

(回忆一下，宽限期可以让客户端平稳地度过 Master 的失效期而不用丢失会话或者是锁，Boxwood 的”grace period”等价于 Chubby 的”session lease”，是另外一个概念)。而且，这种区别是两种系统关于规模及出错概率的不同期望所导致的。尽管 master 的故障恢复很少发生，但是一个 Chubby 锁的丢失对于客户端来说是很昂贵的。

最后，两种系统中的锁是为不同的目的而设计。Chubby 锁是重量级的，需要 sequencer 来保证外部资源的安全，而 Boxwood 的锁是轻量级的，主要是为了在 Boxwood 内部使用。

6. 总结

Chubby 是一个为那些 google 内部的分布式系统的错粒度同步行为设计的分布式锁服务。它已经因名字服务及存储配置信息而广泛使用。

它是设计基于很多已经相互结合地很好的人们所熟知的想法：用于多副本容错的分布式一致性算法，保持了简单语义的用于降低服务端负载的客户端一致性缓存机制，实时性的更新通知，类文件系统接口。我们通过使用缓存机制，协议转换服务器，简化负载来使得单个 Chubby 实例可以扩展到数万个客户端的规模。未来我们还希望通过代理和 partitioning 来增加扩展性。

Chubby 已经成为 Google 首要的内部名字服务；它为很多系统(比如 MapReduce)提供了一种通用的协调机制；存储系统 GFS 和 Bigtable 使用 Chubby 来从多个冗余副本中选举一个 primary；而且它也是那些需要高度可用性的文件的标准存储设施，比如访问控制列表。

7. 致谢

很多人为 Chubby 系统做出了贡献：Sharon Perl 编写了基于 Berkeley DB 的副本层；Tushar Chandra 和 Rober Griesemer 编写了取代 Berkeley DB 的可复制数据库；Ramsey Haddad 将 API 连接到 GFS 接口；Dave Presotto,Sean Owen,Doug Zongker 及 Praveen Tamara 分别编写了 Chubby DNS，Java，以及命名协议转换器，以及完整的 Chubby 代理；Vadim Furman 增加了打开句柄及文件缺失信息的缓存；Rob Pike,Sean Quinlan 和 Sanjay Ghemawat 给了很多有价值的设计建议；很多 google 开发者帮助发现了早期的缺点。

译考文献：

http://blog.xiping.me/2010/12/google-chubby-in-chinese.html#more-854

中文版 Chubby 论文

http://net.pku.edu.cn/~course/cs501/2008/assign/prj/CNDS/CS501-Paper-CNDS.doc

英文版 Chubby 论文

http://www.ibm.com/developerworks/cn/opensource/os-cn-zookeeper/

分布式服务框架 Zookeeper

http://zookeeper.apache.org/doc/r3.3.2/zookeeperOver.html
ZooKeeper: A Distributed Coordination Service for Distributed Applications
http://zookeeper.apache.org/doc/r3.3.2/recipes.html
ZooKeeper Recipes and Solutions

海量数据分析：Sawzall 并行处理

作者：Rob Pike, Sean Dorward, Robert Griesemer, Sean Quinlan Google Inc 2006-01-05
译者：崮山路上走 9 遍 2007
出处：http://sharp838.mblogger.cn http://peopleyun.com/?p=896

[序：Google的工程师为了方便内部人员使用MapReduce，研发了一种名为Sawzall的DSL，同时 Hadoop也推出了类似Sawzall的Pig语言，但在语法上面有一定的区别。今天就给大家贴一下Sawall的论文，值得注意的是其第一作者是UNIX大师之一(Rob Pike)。原文地址，并在这里谢谢译者崮山路上走 9 遍。]

概要

超大量的数据往往会采用一种平面的正则结构，存放于跨越多个计算机的多个磁盘上。这方面的例子包括了电话通话记录，网络日志，web 文档库等等。只要这些超大量的数据集不能装在单个关系数据库里边的时候，传统的数据库技术对于研究这些超大数据集来说那就是没有意义的。此外，对于这些数据集的分析可以使用简单的，便于分布式处理的计算过程来表示：比如过滤，聚合，统计抽取，等等。

我们在这里介绍这样一种这样的自动化分析系统。在过滤阶段，查询请求通过一种全新的编程语言来快速执行，将数据发送到聚合阶段。无论过滤阶段还是聚合阶段都是分布在上百台甚至上千台计算机上执行的。最后会对他们的结果会进行整理并且保存到一个文件。这个系统的设计-包括两个阶段的划分，以及这种新式的编程语言，聚合器的特性-都是在数据和计算分布在很多台机器上的情况下，内嵌使用并行机制的。

1．介绍

有很多数据集或者因为太过巨大，或者因为非常动态，或者就是因为太笨拙了，而不能有效地通过关系数据库进行管理。典型的场景是一组大量无格式文件-有时候是 PB 级的 (petabytes，2 的 50 次方 1,125,899,906,842,624)-分布在多个计算机上的多个磁盘上(图 1)。这些文件都包含了无数的记录，这些记录是通常会通过一个轴来组织，比如通过时间轴或者地理轴进行组织。例如：这堆文件可能包含一个 web 网页仓库，用来构造 internet 搜索引擎的索引系统，或者这堆文件用来记录上千台在线服务器的健康日志，或者用来记录电话呼叫记录或者商业交易日至，网络包记录，web 服务器查询记录，或者高级一点的数据比如卫星图像等等。

但是对这些数据的分析经常可以表示成为简单的操作，远比普通 SQL 查询要简单的操作来完成。举一个例子，我们通常会统计满足某条件的记录数，或者抽取这些记录，或者查询异常记录，或者构造记录中某一个域值的频率柱状图。另一方

面，查询也可能会较为复杂，但是这些查询依旧可以展示成为通过一系列简单查询来完成，这些简单查询都可以简单映射到这些文件的记录集上。

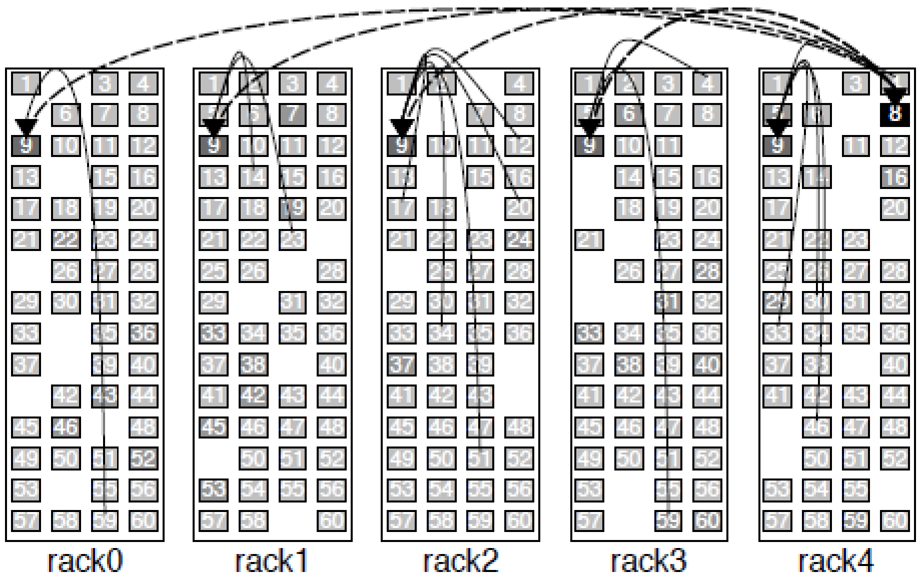


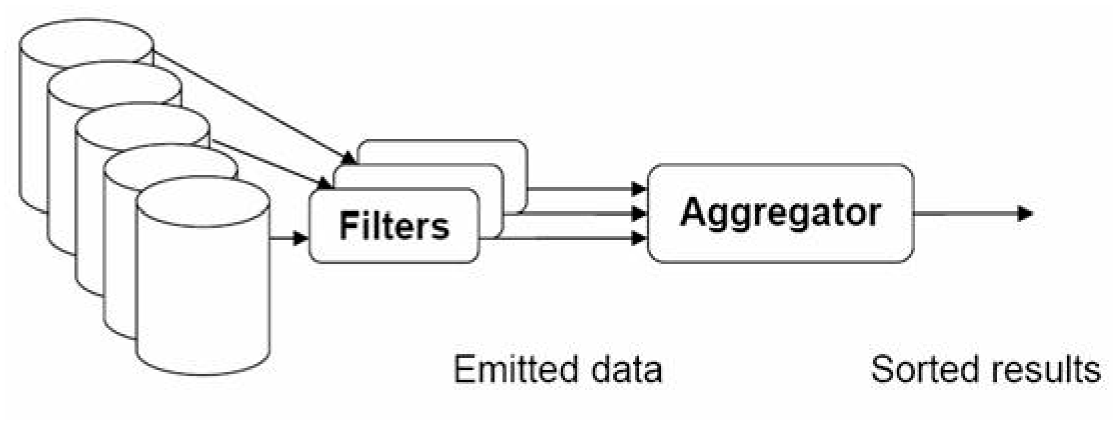
图 1：5 组机架，每组有 50-55 台计算机，每台计算机有 4 个磁盘。这样的一个配置规模可能有到上百TB的待分析数据量。我们可以在所有的 250 多台计算机上分别执行一个过滤阶段来极大的的提高并行度，并且把他们的结果通过网络汇聚到一起（参见图中弧线）。实心弧线代表了从分析机到聚合器的数据流；虚心弧线连接了处于merging状态的聚合数据，起初每个聚合机器上都有一个文件，最终会被收集整理成一个最终文件。

当然，并不是所有的数据都是这样的。某些可能可以直接从传统的编程方法中获益—数字，模式，中间值存储，函数等等—这些由过程性的编程语言如 python 和 awk 都可以提供。如果数据是未结构化的，文本型的，或者该查询需要进行额外的计算以获取记录相关数据，或者在数据集的一次扫描中需要进行很多不同但是相关的计算，一个过程性的语言通常都是一个合适的工具。Awk 是为简化数据挖掘而专门设计的。尽管今天它以被用在很多其他方面，甚至在今天仍然有很多 awk 程序用于分析电话日志。C 和 C++语言也能处理这样的任务，但是易用性要差一些同时需要程序员付出更多的努力。然而 python 和 awk 也不是万能的，比如，它们就不具备在多台计算机上处理数据的内在机制。

由于数据记录存放在多台计算机上，那么用这些计算机本身的能力来进行分析的方法就相当有效。特别是，当单独每一个步骤都可以表示成为每次对独立的记录进行操作的时候，我们就可以把计算分布到所有这些机器上，这样就能达到相当高的吞吐量。（前边提及的每个例子都有这样的特点）。这些简单操作都要求一个聚合的阶段。例如，如果我们统计记录数，我们需要把每一个机器统计出来的记录数相加，作为最终的输出结果。

所以，我们把我们的计算分成两个阶段。第一个阶段我们对每一条记录分别计算，第二个阶段我们聚合这些结果（图 2）。本论文描述的系统更进一步考虑了这个问题。我们用一个全新的编程语言来进行第一个阶段的分析，从处理粒度上，它

一次处理一条记录，并且在阶段 2 严格限制预先定义的处理阶段 1 产出物的聚合器处理的集合。通过约束本模式的计算量，我们可以达到非常高的吞吐量。虽然并非所有的计算都能适合这样的模式，但是仅仅通过不多的代码就能够驱动上千台机器并行计算还是很划算的。



RAW DATA

图 2：总体数据流图，过滤，聚合和比较。每一步都比上一步产生更少的数据。

当然，我们还有很多小问题要解决。计算必须要分解成为小块并且分布到每一个存储数据的节点上进行执行，尽量让计算和数据在一台机器上以避免网络瓶颈。由于使用的机器越多，那么越有可能有机器会在运算中宕机，所以，系统必须要有容错能力。这些都是困难但是有趣的问题，但是他们都必须能够在没有人为干预的情况下完成。Google 有好几个这样的基础架构，包括 GFS[9]和 MapReduce[8]，通过容错技术和可靠性设计来提供了一个非常强大的框架，可以用来实现一个很大的，分布式处理的并行系统。因此我们着重于我们的目标：清晰的表达分析处理，并且迅速执行分析处理。

2. 总览

简要而言，我们的系统通过处理用户提交的用特别设计的编程语言写成的查询，并发的在分布到大量机器上的记录集中，进行记录级别的查询，并且搜集查询结果，通过一组高性能的聚合器进行查询结果的汇聚。这两部分分别执行，通常分布到不同的计算机集群上。

这样的处理典型类型是并发处理分布在成百上千台计算机上的 gigabyte 或者数 Tbyte 数据。一个简单的分析可能需要花去一个 CPU 好几个月的时间，但是通过上千台计算机的并行处理，只需要几个小时的时间就能处理完。

有两个条件决定着系统的设计。首先，如果查询操作是对记录间可交换的，就是说记录处理的先后顺序是不重要的。我们于是可以用任意的顺序来处理这个查询操作。第二，如果聚合操作是可交换的，中间结果的处理顺序是不重要的。此外，如果他们也是可结合的，中间处理结果可以被任意分组或者分成不同的步骤进行

聚合。举一个例子，对于统计数量包括汇总数量来说，无论中间结果如何的累加或者分组结合累加，他们最终的结果都不会受到影响。这个交换性和结合性的约束并不算过分苛刻，他们可以提供很广阔的查寻范围，包括：统计，筛选，取样，柱状图，寻找常见项目，等等。

虽然聚合器组是有限的，但是对于查询阶段来说，应当包括更加通用的内容，我们介绍一种新的解释执行的程序语言 Sawzall[1]（解释语言的性能已经足够了：因为程序多数都是比较小的，而且他们需要处理的数据往往很大，所以往往是受 I/O 的限制，这在性能的章节有所讨论）

一个分析操作如下：首先输入被分解成为要被处理的数据小块，也许是一组独立的文件或者一组记录，这些记录或者文件分布于多个存储节点上。数据小块可以远远多于计算机的数量。

其次，Sawzall 解释器开始处理每一个小块数据。这个处理跨越了大量机器，也许数据和机器绑定在一起，也可能数据在临近的机器上而不在一起。

Sawzall 程序分别处理每一个输入记录。每一个记录的输出结果，可能有 0 个或者多个中间结果值--整数，字串，key-value pairs，tuple 等等--将会与来自其他记录的输出值合并。

这些中间结果于是被发送到运行聚合器的进一步处理的结点上，这些节点比较和减少中间结果，并且构造最终结果。在一个典型的运行中，主要的计算机集群会运行 Sawzall，并且小一点的集群会运行聚合器，这样的结构不仅是体现了计算量的差异，也体现了对网络负载的均衡考虑；每一个步骤，数据流量都比上一个步骤要少（参见图 2）。

当所有的处理都完成之后，结果将被排序，格式化，并且保存到一个文件。

3. 例子

用这个简单的例子可以更清楚的表达这样的想法。我们说我们的输入是一个由浮点数记录组成的文件集合。这个完整的 Sawzall 程序将会读取输入并且产生三个结果：记录数，值得总合，并且值得平方和。

```
count: table sum of int;

total: table sum of float;

sum_of_squares: table sum of float;

x: float=input;

emit count<-1;
```

```
        emit sum<-x;

        emit sum_of_squares <- x*x;
```

前三行定义了聚合器：计数器，和，平方和。关键字 *table*定义了聚合器类型；在Sawzall中，即使聚合器可能是单例的，也叫做table。这些特定的table是属于合计的table；他们把输入的整数或者浮点数的值进行累加。

对于每一个输入的记录，Sawzall初始化预定义的变量 *input*来装载尚未处理的输入记录二进制串。因此，行：

```
      x: float = input;
```

把输入记录从它对外的表示转换成为内嵌的浮点数，并且保存在本地变量 *x*。最后，三个 *emit*语句发送中间结果到聚合器。

当程序执行的时候，程序对每一个输入记录只执行 1 次。程序定义的本地变量每次重新创建，但是程序定义的 table 会在所有执行中共享。处理过的值会通过全局表进行累加。当所有记录都处理了以后，表中的值保存在一个或者多个文件中。

接下来的几节讲述了本系统所基于的部分 Google 的基础架构：协议 buffers，Google 文件系统，工作队列，MapReduce。后续章节描述语言和系统其他部分的细节。

4. 协议 Buffer

虽然在最开始已经考虑了定义服务器之间的消息通讯，Google 的协议 Buffer 也同样用于描述保存在磁盘的持久化存储的记录格式。这个协议 Buffer 的用处很类似 XML，但是要紧凑的多，使用二进制的表示，虽然二进制格式已经是相当紧凑的，但是常常还会在保存到磁盘的时候再进行一个压缩，包裹一个压缩层。

Protocol buffer 使用一个外部的数据描述语言（DataDescription Language DDL）来定义消息内容。协议编译器能够把协议编译成为各种语言的支持代码。协议编译器读取 DDL 描述并且产生用于对数据的:组织,访问,列集及散列处理的代码。编译时候的标志指定了输出的语言：C++，Java，Python,等等。这个产生的代码通过嵌入应用程序中，能够提供对数据记录高效简洁的访问。同时，也应该提供验证和调试保存的协议 buffer 的工具包。

DDL 构造一个清晰，紧凑，可扩展的针对二进制记录的描述，并且对记录的字段进行命名。Protocol buffer 类型十分类似于 C 的结构体；它们由已命名的类型化的域组成。但是,DDL 为每个域提供两个额外的属性:一个可区分的内部标签，用于在二进制表示中标识该域，一个用于表示域是可选的还是必需的。这些额外的属性，允许 Protocol buffer 进行向后兼容的扩展，只要把新添加的域设为可选并且为它们分配一个未使用的标签即可。

比如下面的 Protocol buffer 描述中，具有两个必需的域，域 x 具有 tag 值 1，域 y 具有 tag 值 2.

```
parsed message Point {
required int32 x = 1;
required int32 y = 2;

};
```

为了扩展这个二维的点，可以增加一个具有新的 tag 值的可选域。所有现存的 Point 数据仍然是可读的；它们与新的定义兼容，因为新的域是可选的。

```
parsed message Point {
required int32 x = 1;
required int32 y = 2;
optional string label = 3;

};
```

我们系统操作的大部分数据集都是按照协议 buffer 约定的格式存储的记录。协议编译器通过增加对 Sawzall 的扩展来提供在新语言下的协议 buffer 的高效 IO 性能。

5. Google 文件系统（GFS）

我们系统访问的数据集通常保存在 GFS 内，就是 Goole 的文件系统[9]。GFS 提供一个可靠的分布式存储系统，它可以通过分布在上千台计算机的 64M”块”组织成为上 Petabyte 级别的文件系统。每一个块都有备份，通常是 3 个备份，在不同的计算机节点上，这样 GFS 可以无缝的从磁盘或者计算机的故障中容错。

GFS 是一个应用级别的文件系统，有着传统的分级的命名机制。数据集本身通常用一个常规的结构存放，这些结构存放在很多独立的 GFS 文件中，每一个 GFS 文件大小接近 1G。例如，一个文档仓库（web 搜索引擎的爬虫抓取的结果），包含数十亿 HTMLpages，可能会存放在上千个文件中，每一个文件压缩存放百万级别的文档，每个文档大概有数 K 字节大小。

6. 工作队列和 MapReduce

把工作安排到一组计算机集群上进行工作的处理软件叫做（稍稍有点容易误解）工作队列。工作队列很有效的在一组计算机及其磁盘组上创建了一个大尺度的分时共享机制。它调度任务，分配资源，报告状态，并且汇集结果。

工作队列和 Condor[15]等其他系统比较类似。我们经常把工作队列集群和 GFS 集群部署在相同的计算机集群上。这是因为 GFS 是一个存储系统，CPU 通常负载不太高，在 CPU 空闲阶段可以用来运行工作队列任务。

MapReduce[8]是一个运行在工作队列上的应用程序库。它提供三个主要功能。首先，它提供一个给予大量数据的并行处理的程序运行模式。第二，它把应用从运行在分布式程序的细节中隔离出来，包括类似数据分布，调度，容错等等。最后，当发现条件许可时，各个计算机或者存储自己的 GFS 数据节点上的应用程序可以执行计算，减少网络的流量。

正如 MapReduce 名字本身所代表的含义，这个执行模式分成两个步骤：第一个步骤是把一个执行操作映射到数据集合的全部元素；第二个步骤是简化第一个步骤地输出结果，并且产生最终的应答。例如，一个使用 MapReduce 的排序程序将会映射一个标准的排序算法到数据集合的每一个文件上，接下来就是运行一个合并排序程序来简化第一个步骤出来的单独结果，并且产生最终地输出。在上千台机器的 Cluster 中，一个 MapReduce 程序可以用每秒排序 1G 数据的速度排序上 TB 的数据[9]。

我们的数据处理系统是基于 MapReduce 的最上层的。Sawzall 解释器运行在 Map 阶段。这是在大量机器上并发完成的，每一个执行实例处理一个文件或者一个 GFS 块。Sawzall 程序对每一个数据集的记录只执行一次。Map 阶段的输出是一个数据项的集合，并且是交给聚合器去处理。聚合器在 Reduce 阶段运行来合并结果成为最终的输出结果。

接下来的章节讲述这些处理的细节。

7. Sawzall 语言概览

作为一种查询语言，Sawzall 是一种类型安全的脚本语言。由于 Sawzall 自身处理了很多问题，所以完成相同功能的代码就简化了非常多-与 MapReduce 的 C++ 代码相比简化了 10 倍不止。

Sawzall 语法和表达式很大一部分都是从 C 照搬过来的；包括 for 循环，while 循环，if 语句等等都和 C 里边的很类似。定义部分借鉴了传统 Pascal 的模式：

```
i: int ; # a simple integer declaration;

i: int=0; # a declaration with an initial value;
```

基本类型包括整数（*int*），是 64 位有符号值；浮点数（*float*），是一个double精度的IEEE浮点数；以及很类似整数的 *time*和 *fingerprint*。*time*是毫秒级别的时间，并且函数库包括了对这个类型的转换和操作。*Fingerprint*代表了某个值的内部计算的hash值，可以很容易通过建立数据的 *fingerprint*来构造聚合器索引。

同时，Sawzall也有两种基本的数组类型：*bytes*，类似C的 *unsigned char*的数组；*string*，*string*用来存放UNICODE的字符串。在Sawzall中没有”字符”类型；*byte*数组和 *string*的基本元素是int，而虽然int的容量远比字节或者字符串的基本元素来得大。

复合类型包括数组，maps（本文中，maps 具有自己的含义，与通常意义上的 map 不同），tuples。数组是用整数作为下标检索的，maps 是类似于关联数组或者 Python 字典的类型，可以用任意类型检索，可以根据需要建立无序的索引。最后 tuples 是对数据的任意分组，类似 C 或者 PASCAL 的结构类型。任何类型都可以有一个正式的名字。

类型转换操作是把数据从一种类型转换成为另一种类型，并且 Sawzall 提供了很广泛的类型转换。例如，把一个字符串表示的浮点数转换成为一个浮点数：

```
f: float;

s: string = "1.234";

f = float(s);
```

部分转换是可以带参数的：

```
string(1234, 16)
```

就可以把一个整数转换成为一个 16 进制的字符串。并且：

```
string(utf8_bytes, "UTF-8")
```

转换一个 UTF-8 的 byte 数组成为一个 unicode 字符串。

为了方便起见，并且为了避免某些语言定义上的啰嗦，编译器可以在初始化定义的时候隐含的做适当的转换操作（使用缺省的转换参数）。因此：

```
b: bytes = "Hello, world!\n";
```

等价于显示的转换：

```
b: bytes = bytes("Hello, world!\n", "UTF-8");
```

任何类型的值都可以转换成为字符串，这是为了调试的方便考虑。

Sawzall最重要的转换是和协议buffer相关的。Sawzall有一个编译时刻参数：proto，有点类似C的 *#include*指令，可以从一个定义了Sawzall tuple类型的文件加载DDL协议buffer。通过 tuple描述，就可以转换输入的协议buffer到Sawzall的值了。

对于每一个输入记录，解释器都需要把这个由二进制数组表达的值初始化到特定的输入变量中，尤其是转换到协议 buffer 类型的输入变量中去。Sawzall 程序对于每一个记录的执行都是由下边这条语句隐式执行的：

```
input: bytes = next_record_from_input();
```

因此，如果文件：some_record.proto 包含了类型 Record 的协议 buffer 的定义，那么下边的代码会把每一个输入记录分析到变量 r 中：

```
proto "some_record.proto" # define ’Record’

r: Record = input; # convert input to Record
```

Sawzall 有很多其他的传统特性，比如函数以及一个可以广泛选择的固有的基础函数库。在基础函数库中，有那些给调用代码使用的国际化函数，文档分析函数等等。

7.1. 输入和聚合

虽然在语句级别上，Sawzall 是一个很传统的语言，但是它有两个非常不寻常的特性，都在某种意义上超越了这个语言本身：

1. Sawzall 程序定义了对于数据的单个记录的操作。这个语言没有提供任何可以同时处理多条记录的方法，以及没有提供通过一个输入记录的值来影响另一个记录的方法。
2. 唯一的输出原语是 emit 语句，这个语句发送数据到一个外部的聚合器来汇聚每一个记录的结果并且在聚合器进行结果的加工和处理。

因此普通的 Sawzall 程序行为是使用输入变量，用转换操作把输入的记录分析到一个数据结构，检查数据，并且处理成一些值。我们在第三节可以看到这种模式的一个简单例子。

下边是一个更有代表性的 Sawzall 程序例子。对于给定的我们源代码管理系统的源代码提交记录集合，这个程序会用分钟级别的分辨率，给出周的提交变化频率表。

```
proto "p4stat.proto"

submitsthroughweek: table sum[minute: int] of count: int;

log: P4ChangelistStats = input;

t: time = log.time; # microseconds

minute: int = minuteof(t)+60*(hourof(t)+24*(dayofweek(t)-1));
```

```
emit submitsthroughweek[minute] <- 1;
```

这个程序一开始从文件p4stat.proto引入了协议buffer描述。在这个文件中定义了类型：*P4ChangelistSTats*(程序员必须明确知道这个类型是从proto引入的，而且还要知道这个是由协议buffer DDL定义的)

接下来就是定义了 *submitsthroughweek*。它定义了一个sum值的table，这个table使用一个整数minute作为下标。注意index值在table定义的时候是给出了一个可选的名字（minute）。这个名字没有任何语义，但是使得这个定义更容易理解，并且提供了一个聚合输出的域标签。

log 的定义把输入的 byte 数组转换成为 Sawzall 的类型：P4ChangelistStats，这个转换是用（proto 语句引入的代码转换的），这个类型是 tuple 类型，保存在输入变量 log 里边。接着我们把 time 值取出来，并且接着就保存到变量 t 中。

接下来的定义有着更复杂的初始化表达式，这个表达式使用了一部分内嵌的函数，用来从 time 值来计算基准的周分钟基线数字[2]。

最后，emit 语句通过增加该分钟的数字来统计这个提交情况。

总结一下，这个程序，对于每一个记录，都取得时间戳，把时间转换成为本周的分钟数，然后在这周的对应分钟发生次数增加 1。并且，隐式的，这个会重新取下一个记录进行循环处理。

当我们在全部的提交日志上运行这个程序，这个记录跨越了很多个月，并且输出结果，我们可以看到一个按照分钟区分的聚合的周活动趋势。输出结果可能像这样的：

```
submitsthroughweek[0] = 27

submitsthroughweek[1] = 31

submitsthroughweek[2] = 52

submitsthroughweek[3] = 41

…

submitsthroughweek[10079] = 34
```

当使用图像表达，那么这个图就像图三一样。

我们举这个例子要表达的意思当然不是说这个提交源码的频率数据如何如何，而是说这个程序怎样产生抽取这个数据出来。

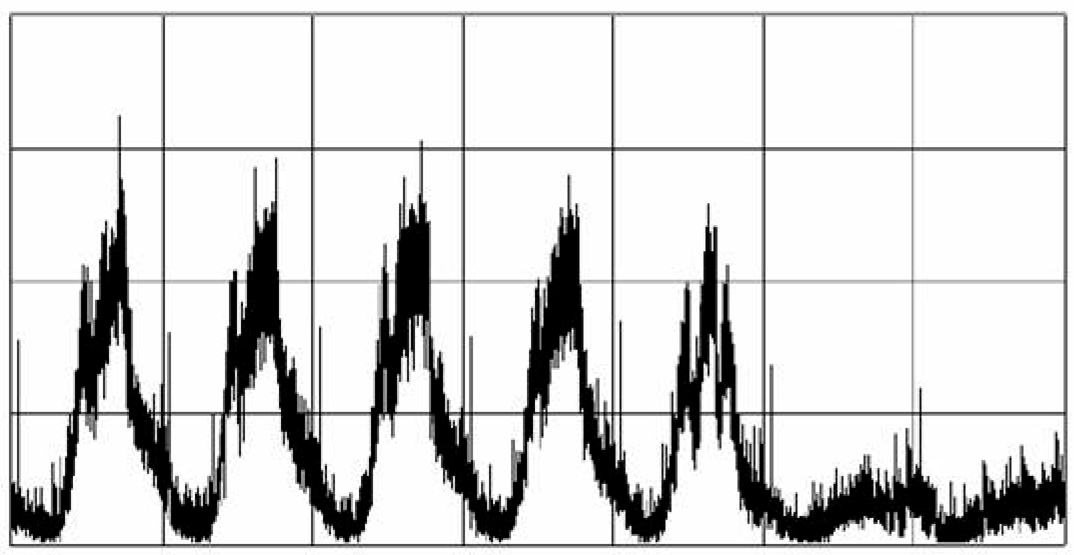


图 3：周源代码提交频率。本图从周一早上凌晨 0 点开始。

7.2. 聚合器补充说明

因为某些原因，我们在本语言之外完成聚合。应该由一个传统的语言来用语言自身的处理能力来处理结果，但是由于聚合的算法可能会相当的复杂，最好用某种本地化的语言来实现。更重要的是，虽然在语言层面上隐藏了并行的机制，但是在过滤阶段和聚合阶段划一条清晰的界限能够有助于更高级别的并行处理。在 Sawzall 中，看不到明显的处理多条记录的痕迹{!通常看到的是它在处理一条记录，记录的迭代过程被隐藏在底层的实现中了}，但是一个典型的 Sawzall job 是在上百或者上千台机器上并发操作上百万条记录，

集中精力在聚合器上可以创造出很不寻常的聚合器。现在已经有许多聚合器；下边是一个不完整的列表：

- 搜集器

```
c: table collection of string;
```

一个简单的输出结果列表，这个结果在列表中是任意顺序的。

- 采样器

```
s: table sample(100) of string
```

类似搜集器，但是存的是无偏差的输出结构的采样值。这个采样的大小是用参数体现的。

- 累加

```
s: table sum of (count:int,revenue:float);
```

所有输出结果的合计。这个输出结果必须是算数的或者可以以算术为基础的（也就是可累加的，by 译者），就像例子中的 tuple 结构那样（也就是说一般可以是 sum of int,也可以像上边说的一样，可以用 sum of (count:int,revenue:float)这样的 tuple 结构。对于复合值，元素是按照内部的项进行累加的。在上边的例子，如果 count 始终为 1，那么平均 revenue 可以在处理完和以后用 revenue 除以 count 来得到。

● 最大值

```
m: table maximum(10) of string weight length: int;
```

取得最大权重的值。每一个值都有一个权重，并且最终选择的值是根据最大权重来选择的。这个参数（例子中是 10）规定了需要保留的最终输出的值数量。权重是以明确的 keyword 来描述的，并且它的类型（这里是 int）是在这里定义的，它的值是 emit 语句给出的。对上边例子来说，emit 语句如下：

```
emit m <- s weight len(s);
```

这样将会在结果中放置最长的字符串。

● 分位数

```
q: table quantile(101) of response_in_ms:int;
```

是用输出的值来构造一个每个概率增量分位数的累计概率分布（算法是一个 Greenwald 和 Khanna 的分布式算法[10]）。这个例子可以用来查看系统的响应变化的分布情况。通过参数 101，这个参数用来计算百分点。第 50 个元素是中间点的响应时间，第 99 个元素是 99%的响应时间都小于等于第 99 个元素。

● 最常见

```
c: table top(10) of language: string;
```

top table 评估这个值是否最常见（与之对应的，maximun table 找到最高权重的值，而不是最常见的值）

例如：

```
emit t <- language_of_document(input);
```

将会从文档库中建立 10 个最常见的语言。对于很大的数据集来说，它可能需要花费过大的代价来找到精确的出现频率的 order，但是可以有很有效的近似算法。top table 是用了 Charikar,Chen,Farach-Colton[5]的分布式算法。算法返回的最常见的频率是极为接近真实的出现频率。因为它的交换性和结合性也不是完全

精确的:改变处理的输入记录先后顺序确实会影响到最终的结果。作为弥补措施，我们在统计元素个数之外，也要统计这些个数的误差。如果这个误差和元素个数相比比较小，那么结果的正确度就比较高，如果错误相对来说比较大，那么结果就比较差。对于分布不均匀的大型数据集来说，top table 工作的很好。但是在少数情况下比如分布均匀{!即各个记录出现的频率相差不大}的情况下，可能会导致工作的不是很成功。

● 取唯一

```
            u: table unique(10000) of string;
```

unique table 是比较特别的。它报告的是提交给他的唯一数据项的估计大小。sum table 可以用来计算数据项的总和个数，但是一个 unique table 会忽略掉重复部分；最终计算输入值集合得大小。unique table 同样特别无论输入的值是什么类型，它的输出总是一个 count。这个参数是给出了内部使用的 table 大小，这个是用来内部作评估是用的内部表；10000 的参数值会让最终结果有 95%的概率正负 2%的误差得到正确的结果(对于 N,标准偏差是大概 N*参数**(-1/2))

7.3. 聚合器实现

有时候也会有新的聚合器出来。虽然聚合器用处很大，但是增加一个新的聚合器还算容易，但是也不是一件简单的事情。聚合器的实现需要与 Sawzall 的运行时环境连接，在系统内部交互，管理底层数据格式及计算分布，甚至需要支持所有解释器所支持的数据类型，这就增加了实现的复杂性。为了简化这项任务，Sawzall 的运行时环境提供了一些支持，但是目前它们还未能像 Sawzall 语言本身那样丰富。

Sawzall 的运行时环境管理着 emit 值的输出管道,负责在机器间传递数据等等。每个聚合器需要实现 5 个函数接口，它们负责处理类型检查，聚合器状态分配，合并(merging)输出值,将数据打包成中间格式以准备更进一步的合并,产生最终输出数据。为增加一个新的聚合器，程序员必须实现这 5 个函数接口，同时将它们链接到 Sawzall 的运行时环境。

对于简单的聚合器，比如 sum，它的实现就很直接。对于更复杂的聚合器，比如分位数和 top，必须注意要选择一个符合交换律和结合律的算法，并且这个算法要在分布式处理上有足够的效率。我们最小的聚合器实现大概只用了 200 行 C++代码，最大的聚合器用了大概 1000 行代码。

有些聚合器可以作为 map 阶段的一部分来处理数据{!实际上就是 Combining}，这样可以降低聚合器的网络带宽。例如 sum table 可以本地作各个元素的累加，只是最后把本部分的最终结果发往远端的聚合器。用 MapReduce 的词语来说，这就是 MapReduce 的合并(Combining)阶段，一种在 map 和 reduce 中间的优化阶段。

7.4. 带索引的聚合器

聚合器可以是带索引的，这个可以使得每一个索引下标的值都有一个单独的聚合器。这个 index 可以是任意的 Sawzall 类型，并且可以是一个聚合器的多维的结构下标。

例如，如果我们检查 web 服务器的 log，table：

```
          table top(1000)[country:string][hour:int] of
request:string;
```

可以用来找到每一个国家每一个小时的最常用的请求字串。

当新的索引值产生的时候，就会动态产生一个独立的聚合器，某种意义上比较类似 map，但是是和所有运行的机器无关。聚合阶段会比较每一个索引下标对应的值，并且产生适当的聚合值给每一个索引值。

作为整理(collation)的一部分，数据值将按照索引排序，这样使得从不同机器上合并最终结果比较容易。当任务完成的时候，输出值就按照索引进行排序了，这就意味着聚合器的输出是索引顺序的。

index 本身就是构造了一个有用的信息。就像上边讲述的 web 服务器的例子，当运行完以后，在 country 索引的记录中就构造了请求接收到的国家集合。另外，index 的引入使得可以用 index 对结果集进行分类。table sum[country:string] of int 产生的索引结果将会等同于去掉重复项以后的 table collection of country:string 的结果值。

8. System Model

下边介绍本语言的基本特性，通过对数据分析的建立，我们可以给出高级别的系统模式概览。

系统运行是基于一个批处理的模式的：用户提交一个工作，这个工作分布在一个固定的文件集合上，并且在执行完成以后搜集输出的结果。输入格式和数据源(通常是文件集）以及输出目标都是在程序语言外指定的，通过执行工作的参数形式来递交给系统进行执行。

负责提交的命令行工具叫做 saw，它有一些基本参数用于定义将要运行的 sawzall 程序，一个用于匹配包含待处理的数据记录的文件名称的模式，一组输出文件集合。一个典型的 job 可能采用如下命令启动：

```
saw --program code.szl \
--workqueue testing \
--input_files /gfs/cluster1/2005-02-0[1-7]/submits.* \
--destination /gfs/cluster2/$USER/output@100
```

program 这个 flag 用于指定将要运行的 Sawzall 程序的源码文件，workqueue 用于指定它将要在哪个工作队列集群上运行，input_files 参数支持使用 Unix shell 风格的文件名匹配元字符来指定输入文件，destination 用于指定产生的输出文件，@用于指定文件数目。

运行完毕之后，用户可以使用一个以 source 为参数的，以上面的 destination 所指定的输出路径为值的工具查看结果：

```
dump --source /gfs/cluster2/$USER/output@100 --format csv
```

该程序负责将分布在各个文件中的结果归并到一块，并以指定的格式打印出来。

当系统接收到一个 Sawzall 的 job 请求，Sawzall 处理器就开始检查这个程序是否语法正确。如果语法正确，源代码就发送给各个需要执行的机器，每一个机器就开始分析代码并且开始执行。执行程序的机器数是由输入大小及输入文件数决定的，默认的上限是一个工作队列中的可用机器数的某个百分比。Saw 工具中有一个 flag 可以用于覆盖默认行为并显示设置这个数字。

每个机器编译这个 Sawzall 程序，然后在自己的输入数据片段上执行它，每次一条记录。输出值被传递给 Sawzall 的运行时系统，运行时系统负责收集它们，为降低需要传送给聚合器机器的数据量会在本地执行一些初始的聚合操作。当中间结果所占用的内存达到了一定的阈值或者输入处理完毕后，运行时系统就发送数据。

聚合器机器(它的个数是由 saw 命令里的 destination 所指定的数字决定的)负责收集来自执行机器的中间输出结果，将它们合并，每个机器产生的输出是一个有序子集。为保证合并过程能够看到所有的必需数据，中间输出结果会根据一个特殊的表格或者它所具有的一个索引值被确定性的发送到特定的聚合器机器{!即 MapReduce 的 hash 过程，这样可以保证相同 key 值的记录会被放到同一台机器上处理}。每个聚合器机器的输出内容会被写入 GFS 上的一个文件，因此整个 job 的数据就是一组文件集合，一个聚合器机器对应一个。因此聚合操作是并行的，可以充分利用集群的资源。

通常生成的输出内容在聚合器机器上的分布相当均匀。表中的每个元素具有相对合适的大小，而且这些元素会分布到不同的机器上。Collection 类型的表是一个例外，这种表可能会变得非常大。为避免单个聚合器机器的过载，在 Collection 表中的值在聚合器机器上以 round-robin 的方式分布。聚合器机器无法看到所有的中间结果，但是根据定义 Collection 表不会合并重复结果，因此这不是一个问题{!即对于 Collection 表，它的相同的 key 值可能分布到不同机器上，但是因为它不需要对相同的 key 值进行合并处理，因此这不会造成问题}。

由聚合器机器所存储的值还不是最终结果，而是保存了一些用于生成最终结果所需信息的中间结果，实际上它与从执行机器传递到聚合器机器的中间结果具有相同的格式。让这些值处于中间结果状态，就可以让它与另一个 job 产生的值进行合并。这样大的 Sawzall job 就可以划分成多个小片来执行，最后通过在最终阶

段将结果合并即可。(与普通的 MapReduce 相比，这是该系统所具有的一个优势；实际中那些运行数天数星期的 MapReduce job，通常可能会出问题)。

{!基于 table 的类型，输入 table 的值可以是最终格式的值，也可以是某种中间结果的值，这些中间结果便于进行合并或者处理。这种合并处理必须能够良好的结合起来才能工作的一个步骤。某些工作由于十分巨大，而结合率允许他们拆成多个小块，并行运行，最后再合并在一起。（这是本系统的一个优势，优于平坦模式的 MapReduce;因为 MapReduce 可能会在一个需要运行几天几周的任务上出问题）}

这种多阶段的合并策略—在执行机器，聚合器机器或者是跨 job 的—这也是聚合器必须是满足结合率才能工作的原因，同时它们也必须满足交换率，因为输入记录的处理顺序是未定义的，同时中间结果的合并顺序也是未知的。

通常，数据的分析结果要比输入小得多，但也有些情况并非如此。例如，我们的程序可以用一个带索引的 collection table 来对数据作多维的组织，在这样的情况下，输出结果与输入差不多大。即使是很多的数据集，这种转换也可以高效的进行。均匀划分的输入和输出独立的穿越执行和聚合引擎。每个聚合引擎会采用基于磁盘的排序对它是输出片段进行排序。这样最终结果就可以通过一个归并排序就可以有效的进行合并以产生最终输出。

8.1 实现

Sawzall 语言通过用 C++编写的传统编译器实现，目标语言是一个可解释的指令集合或者是字节码。编译器及字节码解释器都在同一个二进制文件里，因此用将源代码提交给 Sawzall，系统就可以直接运行它。(这是一个普通但是并不通用的设计，比如 java 将它的编译器和解释性分离成独立的程序)。实际上 Sawzall 语言系统是以库的形式提供，具体一些外部接口负责接受源代码，进行编译及执行，同时绑定了一些外部的聚合器。

聚合器是通过 saw 实现的，saw 是一个链接了 Sawzall 编译器和解释器的程序，同时是实现在 MapReduce 之上，在 Map 阶段运行 Sawzall 程序，Reduce 阶段运行聚合器。MapReduce 负责管理执行阶段和聚合阶段的机器的分布，将计算放置在靠近持有数据的 GFS 节点上以优化网络开销及提高吞吐率，处理机器故障及其他错误。因此系统实际上就是一个“MapReduction”，但不是一个普通的，它实际上包含了两个 MapReduce 实例。

首先，有一个简短的 job 并行运行在工作集群上来计算输入大小并设置主 job。(可能会有数千个输入文件，因此值得并行地去检查它们的大小)。之后，第二个 job 开始运行，saw 实际上扮演了一个对 MapReduce 的简易包装器的角色，它根据输入自动设置 job 运行参数，而不是像通常的那样需要用户描述的 flag 来控制。更重要的是 Sawzall 的方便性及它的聚合器。撇开其简单的编程模型所提供的舒适，Sawzall 可以运行任何提交给它的代码，而其他的 MapReductions 可能是用 C++编写的，还需要单独编译。此外，Sawzall 的聚合器提供的功能显著提高了 MapReduce 的力量。当然可以实现一个包含了 top 这样的聚合器的一个

MapReduce 的包装库，但是它不可能像 Sawzall 这样简单方便。此外，关于聚合器的想法也源自于 Sawzall 所提供的系统模型。尽管有些严格，Sawzall 的执行模型也产生了一些不寻常的创造性的编程思想。

8.2 chaining

Sawzall 中一个常用的输入是把结果数据注入一个传统的关系数据库中，以备后续的分析。通常这些都是有不同的用户程序来注入，也许是用 Python，它把数据转换成为通过 SQL 指令建立的表。我们以后也许会提供更直接的方法来完成把结果注入到数据库的动作。

Sawzall 的结果有时也用于其它 Sawzall 程序的输入，这个就是链式处理。链式处理的简单例子就是精确计算输出的”top 10”列表。Sawzall 的 top table 虽然高效，但是他不精确。如果需要精确的结果，那么就需要分为两个步骤。第一步创建一个带索应的 sum table 来统计输入值得频率；第二个步骤是用一个 maximum table 来选择最常见的频率。这样可能有点笨，但是这种方法依旧是计算多维 table 的非常高效的方法。

9. 例子

这里是另外一个完整的例子，演示了 Sawzall 在实际中如何使用。这里是处理一个 web 文档库，并且产生一个结果：对于每一个 web domain，哪一个 page 有着最高的 Page Rank？粗略来说，哪一个是最多 link 指向的 page？

```
proto “document.proto”

max_pagerank_uri:

table maximun(1)[domain:string] of url:string

weight pagerank:int;

doc: Document = input;

url: string = doc.url;

emit max_pagerank_url[domain(url)] <- url

weight doc.pagerank;
```

protocol buffer 的格式是在”document.proto”中定义的。这个 table 是 max_pagerank_url，并且会纪录每一个索引中最高权重的值。这个索引是 domain，值是 URL，权重势 document 的 PageRank。程序处理输入的纪录，解出 URL，并且执行相关的 emit 语句。它会调用库函数 domain(url)来解出 URL 所对应的

domain，并且使用这个 domain 作为 index，把 URL 作为值，并且用这个 document 对应的 PageRank 作为权重。

当这个程序在一个数据仓库上运行的时候，输出对于大部分site，most-linked 网页是 www.site.com-真是令人惊讶。Acrobat 下载站点是adobe.com的top page，并且连接到banknotes.com的就是连接到连接最多的图库站点，并且 bangkok-th.com是最多引用的夜生活page。

因为是用 Sawzall 能简单表达这样的计算，所以程序是又简洁又优美。即使用了 MapReduce，等价的 C++程序也要好几百行代码。

下边是一个例子，使用了多维索引的聚合器。我们目的是通过检索搜索 log，建立一个查询发起点的世界地图。

```
proto “querylog.proto”

queries_per_degree: table sum[lat:int][lon:int] of int;

log_record: QueryLogProto = input;

loc: Location = locationinfo(log_record.ip);

emit queries_per_degreee[int(loc.lat)][int(loc.lon)] <- 1;
```

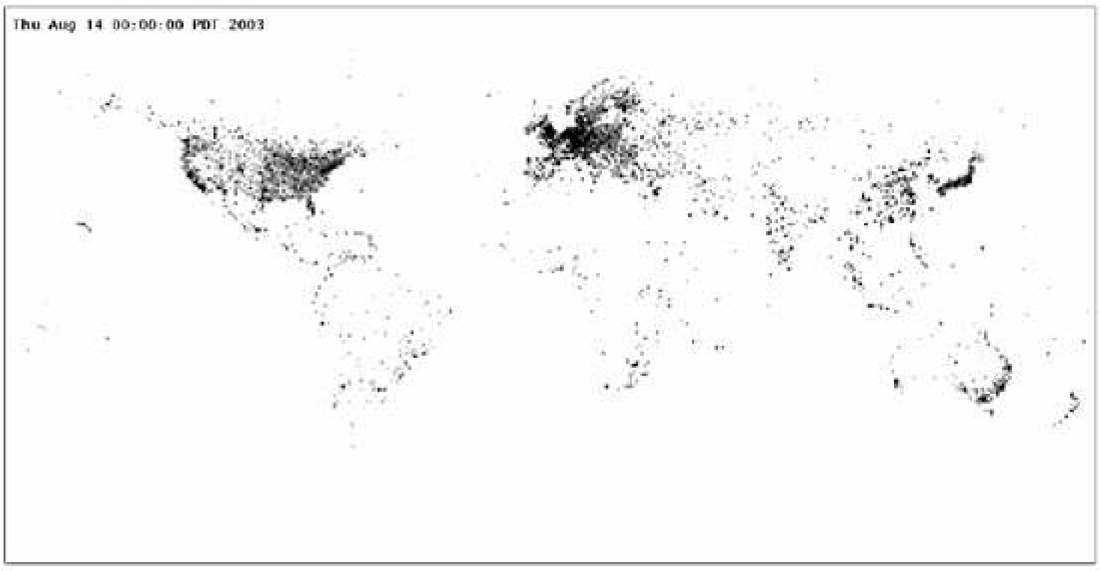
这个程序相当直接，我们引入查询 log 的 DDL，定义一个用了经纬作索引的 table，并且从 log 中解包查询。接着我们是用内嵌函数把这个 IP 地址对应到请求及其的位置（可能是 ISP 的位置），并且为每一个经纬点增加 1。int(loc.lat)把 loc.lat，一个浮点值转换成为一个整数，截断成为一个维数下标。对于高分辨的地图来说，可能要求更精细的计算。

这个程序的输出是一个数组，可以用来构造一个地图，参见图 4。

10. 执行模式

在语句级别，Sawzall 是一个常规的语言，但是从更高的角度看，它有一些特点，所有的设计目的都是为了并行计算。

当然，最重要的是，一次处理一个纪录。这就意味着，其他纪录的处理将不消耗额外的内存（除了在语言本身外把结果提交给聚合器）。Sawzall 在上千台机器上并行执行，是 Sawzall 的一个设计目的，并且系统要求这些机器之间没有额外的通讯。唯一的通讯就是从 Sawzall 的执行结果下载到聚合器。



图四：查询分布

为了强调这点，我们用计算输入记录数的数量来入手。就像我们之前看到的这个程序：

```
count: table sum of int;

emit count <- 1;
```

这个程序将完成统计记录数的工作。与之对比的是，如下的一个错误的程序：

```
count: int = 0;

count ++;
```

这个程序将不能统计记录数，因为，对于每一个记录来说，count 都被设置成为 0，然后再++，最后结果就扔掉了。当然，并行到大量机器上执行，扔掉 count 的效率当然很高。

在处理每一个记录之前，Sawzall 程序都会回到初始的状态。类似的，处理完成一条记录，并且提交了所有的相关的数据给聚合器后，任何执行过程中使用到的资源—变量—临时空间等等—都可以被废弃。Sawzall 因此使用的是一个 arena allocator[11]（单向递增分配，场地分配策略，就是说，从一个内存池中通过单向增加一个指针的方式来分配内存，类似零散内存的管理方式）。当一个记录都处理完成之后，就释放到初始状态。

在某些情况下，重新初始化是不需要的。例如，我们可能会创建一个很大的数组或者影射表来对每条记录进行分析。为了避免对每条记录都作这样的初始化，Sawzall 有一个保留字 static 可以确保这个变量只初始化一次，并且是在处理每条记录的最开始的初始化的时候执行。这就是一个例子：

```
static CJK: map[string] of string = {
“zh” : “Chinese”,
“jp”:”Japanese”,
“ko”,”Korean”,
};
```

CJK 变量会在初始化的时候创建，并且作为处理每条记录的初始化的时候都保留 CJK 变量的值。

Sawzall 没有引用类型；它是纯粹值语义的。数组和 maps 也可以作为值来使用（实现的时候,在大部分情况下,用使用引用计数的 copy-on-write 来提高效率)。某些时候这个比较笨拙-在一个函数中修改一个数组，那么这个函数必须返回这个数组--但是在典型的 Sawzall 程序中，这个并没有太大的影响。但是这样的好处，就可以使得并发处理记录的时候，不需要担心同步问题或者担心交叉使用的问题，虽然实现上很少会用到这个情况。

11. 语言的 Domain 相关特性

为了解决 domain 操作的问题，Sawzall 有许多 domain 相关的特性。有一部分已经讨论过了，本节讨论的是剩下的一部分。

首先，跟大部分”小语言”[2]所不同，Sawzall 是一个静态类型语言。主要是为了可靠性的考虑。Sawzall 程序在一次运行中，会使用数小时，乃至好几个月的 CPU 时间，一个迟绑定（late-arising）动态类型错误导致的代价就有可能太大。另外，还有一个潜在的原因，聚合器使用完整的 Sawzall 类型，静态类型会让聚合器的实现比较容易。类似的争议也在分析输入协议 buffer 上；静态类型可以精确检测输入的类型。同样的，也会因为避免了运行时刻动态类型检测而提高整个的性能。最后，编译时类型检查和强制类型转换要求程序员精确的指出类型转换。唯一的例外是在变量初始化的时候，但是就算在这个时候，类型已经是清晰而且程序也是类型安全的。

从另外的角度上看，强类型保证了变量的类型一定可知，在初始化的时候容易处理。例如：

```
t: time=”Apr 1 12:00:00 PST 2005”;
```

这样的初始化就很容易理解，而且还是类型安全的。并且有一些基本类型的属性也是主机相关的。比如处理 log 记录的 time stamps 的时候，这个 time 基本类型就是依赖于 log 记录的 time stamps 的；对于它来说，如果要支持夏令时的时

间处理就太过奢侈了。更重要的是（近来比较少见了），这个语言定义了用 UNICODE 表示 string，而不是处理一组扩展字符集编码的变量。

由于处理大量数据集的需要，我们有赋予这个语言两个特性：处理未定义的值，处理逻辑量词。下两节详细描述这个特性。

11.1 未定义的值

Sawzall 没有提供任何形式的异常处理机制。相反，他有自己的未定义值的处理概念，用来展示错误的或者不确定的结果，包括除 0 错，类型转换错误，I/O 错误，等等。如果程序在初始化以外的地方，尝试去读一个未定义的值，它会崩溃掉，并且报告一个失败。

def（）断言，用于检测这个值是否一定定义了；如果这个值是一个确定值，他返回 true，否则返回 false。他的通常用法如下：

```
v: Value = maybe_undefined();

if (def(v)) {

compute(v);

}
```

下面是一个必须处理未定义值得例子。我们在 query-mapping 程序中扩展一个时间轴。原始程序使用函数 locationinfo()来通过外部数据库判定 IP 地址的位置。当 IP 地址不能在数据库中找到的时候，这个程序是不稳定的。在这种情况下，locationinfo()函数返回的是一个不确定的值，我们可以通过使用 def()断言来防止这样的情况。

下边就是一个简单的扩展：

```
proto "querylog.proto"

static RESOLUTION: int = 5; # minutes; must be divisor of 60

log_record: QueryLogProto = input;

queries_per_degree: table sum[t: time][lat: int][lon: int] of
int;

loc: Location = locationinfo(log_record.ip);

if (def(loc)) {

t: time = log_record.time_usec;
```

```
            m: int = minuteof(t); # within the hour

            m = m  -  m % RESOLUTION;

            t = trunctohour(t) + time(m * int(MINUTE));

            emit queries_per_degree[t][int(loc.lat)][int(loc.lon)] <-
1;

            }
```

（注意，我们只是简单的扔掉我们不知道的位置，在这里是一个简单的处理）。在 if 后边的语句中，我们用了一些基本的内嵌函数（内嵌常数：MINUTE），来截断记录中的 time stamp 的微秒部分，整理成 5 分钟时间段。

这样，给定的查询 log 记录会扩展一个时间轴，这个程序会把数据构造多一个时间轴，这样我们可以构造一个动画来展示如何随着时间变化而查询位置有变化。

有经验的程序员会使用 def()来保护常规错误，但是，有时候错误混杂起来会很怪异，导致程序员很难事先考虑。如果程序处理的事 TB 级别的数据，一般都会有一些数据不够规则；往往数据集的数据规则度超乎作分析程序的人的控制，或者包含偶尔当前分析不支持的数据。在 Sawzall 程序处理的情况下，通常对于这些异常数据，简单丢弃掉是最安全的。

Sawzall 因此提供了一种模式，通过 run-time flag 的设置，可以改变未定义值的处理行为。通常，如果遇到一个未定义的值（就是说没有用 def()来检测一下），将会终止程序并且会给出一个错误报告。当 run-time flag 设置了，那么，Sawzall 简单的取消这个未定义的值相关的语句的执行。对于一个损坏的记录来说，就意味着对临时从程序处理中去除一样的效果。当这种情况发生的时候，run-time 会把这个作为日志，在一个特别的预先定义的 Collection table 中记录。当运行结束的时候，用户可以检查错误率是否可以接受。对于这个 flag 的用法来说，还可以关掉这个 flag 用于调试-否则就看不到 bug！-但是如果在一个大数据集上运行的时候，还是打开为妙，免得程序被异常数据所终止。

设置 run-time flag 的方法是不太常见的错误处理方法，但是在实际中非常有用。这个点子是和 Rinard etal[14]在 gcc 编译器生成容错代码有点类似。在这样的编译器，如果程序访问超过数组下表的索引，那么生成的代码可以使得程序能够继续执行。这个特定的处理方式参考了 web 服务器的容错设计的模式，包括 web 服务器面临恶意攻击的健壮性的设计。Sawzall 的未定义值的处理增加了类似的健壮性设计级别。

11.2 量词

虽然 Sawzall 是基于单个记录的操作，这些记录可能会包含数组或者结构，并且这些数组或者结构需要作为单个记录进行分析和处理。哪个数组元素有这个值？

所有值都符合条件？为了使得这些容易表达，Sawzall 提供了逻辑量词操作，一组特定的符号，类似”for each”,”for any”,”for all”量词。

在 when 语句的这种特定的构造中，可以定义一个量词，一个变量，和一个使用这个变量的布尔类型的条件。如果条件满足，那么就执行相关的语句。量词变量就像普通的 integer 变量，但是它的基础类型（通常是 int）会有一个量词前缀。比如，给定数组 a，语句：

```
when(i: some int; B(A[i]))

F(i);
```

就会当且仅当对于一些 i 的取值，布尔表达式 B(a[i])为 TRUE 的情况下，执行 F（i）。当 F（i）执行了，他会被绑订到满足条件的值。对于一个 when 语句的执行来说，要求有求值范围的一个限制条件；在这个例子中，隐式的指出了关于数组的下标就是求值的范围。在系统内部实现上，如果需要，那么系统使用 def() 操作来检查边界。

一共有三个量词类型：some，当有任意值满足条件的时候执行（如果超过一个满足条件，那么就任选一个）;each,执行所有满足条件的值；all，当所有的值都满足条件的时候执行（并且不绑定值到语句体）。

when 语句可能包含多个量词，通常可能会导致逻辑编程的混淆[6]。Sawzall 对量词的定义已经足够严格了，在实际运用中也不会有大问题。同样的，当多重变量出现的时候，Sawzall 规定他们将按照他们定义的顺序进行绑定，这样可以让程序员有一定的控制能力，并且避免极端的情况。

下边是一些例子。第一个测试两个数组是否共享一个公共的元素：

```
when(i, j: some int; a1[i] == a2[j]) {

…

}
```

第二个例子扩展了这个用法。使用数组限制，在数组的下标中使用用:符号来限制，他测试两个数组中，是否共享同样的 3 个或者更多元素的子数组：

```
when(i0, i1, j0, j1: some int; a[i0:i1] == b[j0:j1] &&i1 >=
i0+3) {

…

}
```

在类似这样的测试中，不用写处理边界条件的代码。即使数组小于三个元素，这个语句依旧可以正确执行，when 语句的求值可以确保安全的执行。

原则上，when 语句的求值处理是可以并行计算的，但是我们还没有研究这方面的内容。

12 性能

虽然 Sawzall 是解释执行的，但是这不是影响性能的主要因素。大部分 Sawzall 程序都只会带来很少一点的处理开销和 I/O 开销，而大部分的 CPU 时间都用于各种 run-time 的操作，比如分析 protocol buffer 等等。

不过，为了比较单 CPU 的 Sawzall 和其他解释语言的解释执行性能，我们写了一些小的测试程序。第一个是计算 Mandelbrot 的值，来测试基本的算术和循环性能。第二个测试函数用递归函数来计算头 35 个菲波纳契级数。我们在一个 2.8G x86 台式机上执行的测试。表 1 是测试结果，显示了 Sawzall 远比 Python, Ruby 或者 Perl 快，起码这些 benchmarks 上要快。另一方面，在这些测试上，Sawzall 比解释执行的 Java 慢 1.6 倍，比编译执行的 Java 慢 21 倍，比 C++编译的慢 51 倍。

	Sawzall	Python	Ruby	Perl
Mandelbrot runtime factor	12.09s 1.00	45.42s 3.75	73.59s 6.09	38.68s 3.20
Fibonacci runtime factor	11.73s 1.00	38.12s 3.24	47.18s 4.02	75.73s 6.46

表 1：Microbenchmarks. 第一个 Mandelbrotset 计算：500×500 图像，每点最多 500 次叠代。第二个用递归函数计算头 35 个菲波纳契级数。

这个系统的性能关键并非是单个机器上的性能，而是这个性能在处理大数据量时，增加机器的时候性能增加曲线。我们使用了一个 450GB 的压缩后的查询 log 数据，并且在其上运行一个 Sawzall 程序来统计某一个词出现的频率。这个程序的核心代码是类似这样的：

```
result: table sum[key: string][month: int][day: int] of int;

static keywords: array of string =

{ "hitchhiker", "benedict", "vytorin",
"itanium", "aardvark" };

querywords: array of string = words_from_query();

month: int = month_of_query();

day: int = day_of_query();

when (i: each int; j: some int; querywords[i] == keywords[j])

emit result[keywords[j]][month][day] <- 1;
```

我们在 50 到 600 台 2.4G Xeon 服务器上执行了这个测试程序。测试的时间结果在图 5 体现了。在 600 台机器的时候，汇聚器大概可以每秒处理 1.06G 压缩后的数据，或者 3.2G 未压缩的数据。如果这个性能扩展能力是比较完美的，那么随着机器的增加处理性能能近似线形增长，这就是说，每增加一台机器，都能增加一台机器的完整处理性能。在我们的测试中，增加 1 台机器的效率增加大约是相当于增加 0.98 台机器。

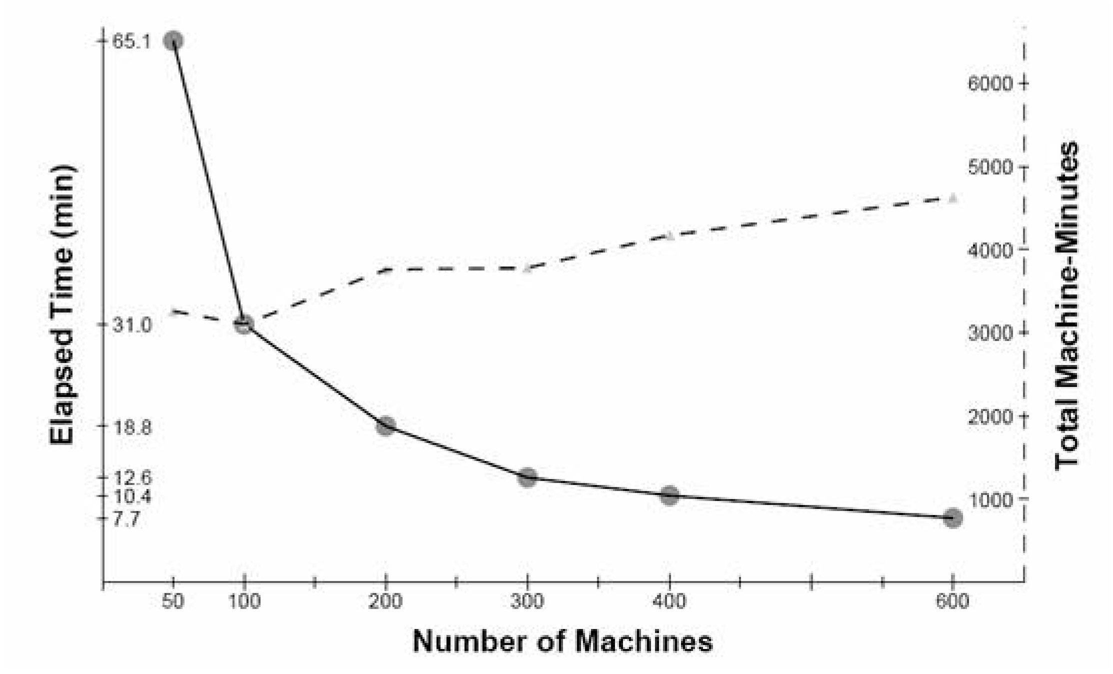


图 5：当增加机器的时候性能变化曲线。实线是花费的时间，虚线是机器的工作时间产出。从 50 到 600 台机器的一个区间内，单机的性能产出仅仅下降了 30%。

为什么需要一个新语言？

为什么我们需要在 MapReduce 之上增加一个新的语言？MapReduce 已经很高效了；还少什么吗？为什么需要一个全新的语言？为什么不在 MapReduce 之上使用现成的语言比如 Python？

这里给出了构造一个特殊目的语言的常见原因。为某一个问题领域构造特定的符号描述有助于程序清晰化，并且更紧凑，更有效率。在语言内嵌聚合器（包括在运行时刻内嵌聚合器）意味着程序员可以不用自己实现一个，这点不像使用 MapReduce 需要自己实现。同样的，它也更符合大规模并发处理超大数据集时候的处理思路，并且根据这个处理思路写出一流的程序。同样的，对协议栈 buffer 的支持，并且提供了平台相关的类型支持，在较低层面上简化了程序开发。总的来说，Sawzall 程序要比基于 MapReduce 的 C++小上 10~20 倍，并且更容易书写。

定制语言还有其他优势包括了增加平台相关的特性，定制的调试和模型界面，等等。

不过，制作这个 Sawzall 的原始动机完全不同：并行，拆分聚合器，并且提供不需要对记录内部作分析就可以最大程度的对记录进行并行处理。它也提供了一个分布式处理的模式，激励用户从另外的思维角度考察并行问题。在现成的语言中比如 Awk[12],Python[1],用户可能要用这个语言书写聚合器，这就可能比较难以做到并行化处理。甚至就算在这些语言中提供了清晰的聚合器接口和函数库，经验老到的用户还有可能要实现他们自己的内容，用以大幅度提高处理性能。

Sawzall 采用的模式已经被证明非常有效。虽然对于少数问题来说，这样的模式还不能有效处理，但是大部分海量数据的处理来说都已经很适用了，并且可以简单用程序实现，这就使得 Sawzall 成为 google 中很受欢迎的语言。

这个语言对用户编程方面的限制也带来额外的一些好处。因为用户程序的数据流是强类型化的，它很容易用来提供记录中的单独字段的访问控制。就是说，系统可以自动并且安全的在用户程序外增加一层，这个层本身也是由 Sawzall 实现的，它用来隐藏敏感信息。例如，产品工程师可以在不被授权业务信息的情况下，访问性能和监控信息数据。这个会在单独的论文中阐述。

14 工具

虽然 Sawzall 仅仅发布了 18 个月，他已经成为了 google 应用最广泛的语言之一。在我们的源码控制系统内已经有上千个 Sawzall 程序（虽然，天生这些程序就是短小精干的）。

Sawzall 工具的一个衡量指标就是它所处理的数据量。我们监控了 2005 年 3 月的使用情况。在 3 月份，在一个有 1500 个 XeonCPU 的工作队列集群上，启动了 32580 个 Sawzall job，平均每个使用 220 台机器。在这些作业中，产生了 18636 个失败（应用失败，网络失败，系统 crash 等等），导致重新运行作业的一部分。所有作业读取了大约 3.2×10^15 字节的数据（2.8PB），写了 9.9×10^12 字节（9.3TB）（显示了”数据合并”有些作用）。平均作业处理大概 100GB 数据，这些作业总共大约等于一个机器整整工作一个世纪的工作。

15 相关工作

传统的数据处理方式通常是通过关系数据库保存数据，并且通过 SQL 查询来进行查询。我们的系统有比较大的不同。首先，数据集通常过于巨大，不能放在关系型数据库里；而且文件直接在各个存储节点进行处理，而不用导入一个数据库服务器。同样的，我们的系统也没有预先设定的 table 或者索引；我们用构造一个特别的 table 和索引来进行这样的相关计算。

Sawzall 和 SQL 完全不同，把高效的处理单个记录分析结果的聚合器接口结合到传统的过程语言。SQL 有很高效的数据库 join 操作，但是 Sawzall 却不行。但是另一方面来说，Sawzall 可以在上千台机器上运行处理超大数据集。

Brook[3]是另一个数据处理语言，特别适合图像处理。虽然在不同的应用领域，就像 Sawzall 一样，它也是基于一次处理一个元素的计算模式，来进行并行处理，并且通过一个聚合器内核来合并（reduce）输出。

另外一种处理大数据的方式是通过数据流的模式。这样的系统是处理数据流的输入，他们的操作是基于输入记录的顺序。比如，Aurora[4]就是一个流模式处理系统，支持单向数据流输入的数据集处理。就像 Sawzall 预定义的聚合器，Aurora 提供了一个很小的，固定操作功能集合，两者都是通过用户定义的函数来体现的。这些操作功能可以构造很多有意思的查询。同 Sawzall 不同的是，部分 Aurora 操作功能是基于输入值得连续的序列，或者输入值得一个数据窗。Aurora 只保存被处理的有限的一部分数据，并且不是为了查询超大的归档库设计的。虽然对 Aurora 来说，增加新的查询很容易，但是他们只能在最近的数据上进行操作。Aurora 和 Sawzall 不同，Aurora 是通过精心设计的运行时刻系统和查询优化器来保证性能，而 Sawzall 是通过强力的并行处理能力来保证性能。

另一种流模式处理系统是 Hancock[7]，对流模式的处理方式进行了扩展，提供了对每个查询的中间状态作保存。这个和 Sawzall 就完全不同，Sawzall 完全不考虑每个输入记录的处理后的状态。Hancock 和 Aurora 一样，专注于依靠提高单进程处理效率，而不是依靠大规模并行处理来提高性能。

16 展望

成百台机器并行处理的生产力是非常大的。因为 Sawzall 是一个大小适度的语言，用它写的程序通常比较小，并且是绑定 I/O 的。因此，虽然它是一个解释语言，实现上效率也足够了。但是，有些超大的，或者超复杂的分析可能需要编译成为机器码。那么编译器需要每台机器上执行一次，然后就可以用这些高速的二进制代码处理每条输入记录了。

有时候，程序在处理记录的时候需要查询外部数据库。虽然我们已经提供了对一些小型数据库的支持，比如什么 IP 地址信息之类的，我们的系统还是可以用一个接口来操作一个外部数据库。因为 Sawzall 对每条记录来说是单独处理的，所以当进行外部数据库操作的时候，系统会暂时停顿，当操作完成，继续处理记录。在这个处理过程中，当然有并行处理的可能。

有时候，我们对数据的分析需要多次处理，无论多次 Sawzall 处理或者从其他系统的处理而导致的多次 Sawzall 处理，比如从一个传统数据库来的，或者一个其他语言写的程序来的；由于 Sawzall 并不直接支持”chaining”(链式处理)，所以，这些多重处理的程序很难在 Sawzall 中展示。所以，对这个进行语言方面的扩展，可以使得将来能够简单的表达对数据进行多次处理，就如同聚合器的扩展允许直接输出到外部系统一样。

某些分析需要联合从不同的输入源的数据进行分析，通常这些数据是在一次 Sawzall 处理或者两次 Sawzall 处理之后进行联合分析。Sawzall 是支持这样的联合的，但是通常要求额外的链接步骤。如果有更直接的 join 支持会简化这样的设计。

更激进的系统模式可以完全消除这种批处理的模式。在激进的模式下，一个任务比如性能检测任务，这个 Sawzall 程序会持续的处理输入数据，并且聚合器跟进这个数据流。聚合器本身在一些在线服务器上运行，并且可以在任何时候来查询任何 table 或者 table 条目的值。这种模式和流式数据库[4][7]比较类似，事实上这个也是基于数据流模式考虑的。不过，在研究这种模式以前，由 Dean 和 Ghemawat 构造的 MapReduce 库由于已经非常有效了，所以这样的模式还没有实现过。也许有一天我们会回到这样的模式下。

17 结束语

随着问题的增大，就需要有新的解决方案。为了更有效的解决海量数据集的大规模并发分析计算，就需要进一步限制编程模式来确保高并发能力。并且还要求不影响这样的并发模式下的展示/应用/扩展能力。

我们的方法是引入了一个全新的语言叫做 Sawzall。这种语言通过强制程序员每次考虑一条记录的方式来实现这样的编程模式，并且提供了一组强力的接口，这些接口属于常用的数据处理和数据合并聚合器。为了能方便写出能并发运行在上千台计算机上执行的简洁有效的程序，学一下这个新的语言还是很超值的。并且尤其重要的是，用户不用学习并发编程，本语言和底层架构解决了全部的并发细节。

虽然看起来在一个高效环境下使用解释语言有点夸张，但是我们发现 CPU 时间并不是瓶颈，语言明确指出，绝大部分程序都是小型的程序，并且大量的时间都耗费在 I/O 上以及 run-time 的本地代码。此外，解释语言所带来的扩展性是比较强大的，在语言级别和在多机分布式计算上的表达都是容易证明扩展能力。

也许对我们系统的终极测试就是扩展能力。我们发现随着机器的增加，性能增长是近似线性增长的。对于海量数据来说，能通过增加机器设备就能取得极高的处理性能。

18 致谢

Geeta Chaudhry 写了第一个强大的 Sawzall 程序，并且给出了超强建议。Amit Pate,Paul Haahr，Greg Rae 作为最早的用户给与了很多帮助。Paul Haahr 创建了 PageRank 例子。Dick Sites， Ren’ee French 对于图示有贡献。此外 Dan Bentley,Dave Hanson,John Lamping,Dick Sites,Tom Szymanski, Deborah A. Wallach 对本论文也有贡献。

19 参考资料

[1] David M. Beazley, Python Essential Reference, New Riders, Indianapolis, 2000.

[2] Jon Bentley, Programming Pearls, CACM August 1986 v 29 n 8 pp. 711-721.

[3] Ian Buck et al., Brook for GPUs: Stream Computing on Graphics Hardware, Proc. SIGGRAPH,Los Angeles, 2004.

[4] Don Carney et al., Monitoring Streams – A New Class of Data Management Applications, Brown Computer Science Technical Report TR-CS-02-04. At http://www.cs.brown.edu/research/aurora/aurora tr.pdf.

[5] M. Charikar, K. Chen, and M. Farach-Colton, Finding frequent items in data streams, Proc 29th Intl. Colloq. on Automata, Languages and Programming, 2002.

[6] W. F. Clocksin and C. S. Mellish, Programming in Prolog, Springer, 1994.

[7] Cortes et al., Hancock: A Language for Extracting Signatures from Data Streams, Proc. Sixth International Conference on Knowledge Discovery and Data Mining, Boston, 2000, pp. 9-17.

[8] Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, Proc 6th Symposium on Operating Systems Design and Implementation, San Francisco, 2004, pages 137-149.

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, The Google File System, Proc. 19th Symposium on Operating System Principles, Lake George, New York, 2003, pp. 29-43.

[10] M. Greenwald and S. Khanna, Space-efficient online computation of quantile summaries, Proc. SIGMOD, Santa Barbara, CA, May 2001, pp. 58-66.

[11] David R. Hanson, Fast allocation and deallocation of memory based on object lifetimes. Software - Practice and Experience, 20(1):512, January 1990.

[12] Brian Kernighan, Peter Weinberger, and Alfred Aho, The AWK Programming Language, Addison-Wesley, Massachusetts, 1988.

[13] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd, The pagerank citation algorithm: bringing order to the web, Proc. of the Seventh conference on the World Wide Web, Brisbane, Australia, April 1998.

[14] Martin Rinard et al., Enhancing Server Reliability Through Failure-Oblivious Computing, Proc. Sixth Symposium on Operating Systems Design and Implementation, San Francisco, 2004, pp. 303-316.

[15] Douglas Thain, Todd Tannenbaum, and Miron Livny, Distributed computing in practice: The Condor experience, Concurrency and Computation: Practice and Experience, 2004.

Case Study GFS： Evolution on Fast-forward

作者：Kirk McKusick & Sean Quinlan
译者：phylips@bmy 2011-8
出处：http://duanple.blog.163.com/blog/static/7097176720117611327190/

[一个 Kirk McKusick 与 Sean Quinlan 之间关于 GFS 的起源和演化的讨论]

在 Google 的早期开发阶段，最初的想法并没有包含一个构建新的文件系统的计划。工作依然是通过公司最早版本的爬虫和索引系统来完成，但是，事情对于核心工程师们很快变得明朗起来，除了构建一个新的系统外他们别无选择，于是 GFS(Google File System)就诞生了。

首先，由于 Google 的目标是要通过使用很多廉价的商品化硬件来构建一个大规模存储网络。因此它必须要假设组件失败是一种常态—这就意味着常规性的监控，错误检测，容错，自动恢复必须作为系统内完整的一部分。而且，即使是 Google 最早期的一些估算，该文件系统的吞吐率需求在任何人看来都是非常可观的—处理 multi-gigabyte 级别的文件，数据集可能包含数 TB 的数据及数百万个对象。很明显，这意味着必须要重新审视关于 IO 操作及块大小的传统假设。同时还需要关注可扩展性。这肯定是一个需要全然不同的扩展性的文件系统。当然，在早期的那些日子里，没有人能说出到底需要什么样的扩展性。但是很快他们就会明白。

虽然已过了近十年，但是 Google 那令人叹为观止的数据存储以及不断增长的应用仍然是建立在 GFS 之上。在这个过程中，该文件系统进行过很多变更—同时那些使用 GFS 的应用程序也不断进行着各种改变—这使得一切成为可能。

为了探索某些关键的初始设计决定以及从那时起所进行的一些改进的缘由，ACM 邀请了 Sean Quinlan 来揭开那些处于不断变更中的文件系统需求以及那些不断演化的思想的神秘面纱。在很多年中，Sean Quinlan 一直是 GFS 的技术领导人，目前已是 Google 的首席工程师，因此他是一个引导我们透视 GFS 的不二人选。为避免局限于 Google 的视野，ACM 又邀请了 Kirk McKusick 来推动这个讨论。他因为在 BSD Unix 上的工作，包括 Berkeley FFS(Fast File System)的原始设计而为人熟知。

这个讨论以初始 GFS 实现中的单 master 设计决定作为开端。乍看，单个的中央 Master 会成为带宽瓶颈—或者，更糟的是可能产生单点失败—但是实际上呢，对于这个决定，Google 的工程师们有他们自己的设计考虑。
■
■
MCKUSICK 原始的 GFS 架构中有一个很有趣也很重要的设计决定就是基于一个单 master。能否解释一下是什么原因导致了这个决定？

QUINLAN 采用单 master 的决定实际上是最初的几个决定中的一个，最主要的是

为了简化总体的设计。也就是说，直接建立一个分布式的 master 当时看来很困难而且会花太多时间。而且，通过这种单 master 策略，工程师也可以简化很多问题。这就有一个中央位置可以用来控制 relication，垃圾回收和很多其他行为，这比在一个分布式的环境上更简单。因此，最后决定将这些集中到一台机器上。

MCKUSICK 那是不是主要是因为这样就可以在一个尽量短的时间内解决很多问题。

QUINLAN 是的。曾经参与 GFS 的一些工程师后来继续去实现 BigTable，一个分布式的存储系统，这个系统就花了好多年。这种将最初的 GFS 建立在单 master 上的决定大大加快了其实现进度。
而且，考察了面临的使用场景之后，当时看起来单 master 的设计也不会引起大的问题。当时所考虑的规模大概是在数百 TB 数据以及数百万文件数。事实上，这个系统一开始工作的很好。

MCKUSICK 但是，之后呢？

QUINLAN 随着底层存储大小的增长，一些问题开始暴露出来。当从数百 TB 上升到 PB 级别，再到数 10PB……这会导致 master 需要维护的元数据数量产生了相应规模的增长。而且，那些扫描元数据的操作也是随数据量线性增长的。这种需要 master 所做的工作大幅增长，所需要存储的数据量也随之增长。
另外，这对于客户端也会是一个瓶颈，即使客户端本身只产生很少的元数据操作—比如，客户端进行一个 open 操作时会与 master 通信。当有成千上万个客户端同时与 master 通信时，假设 master 每秒只能处理几千个操作，那么客户端就不能在一秒内产生这么多请求。而且需要注意的是有很多像 MapReduce 这样的应用程序，可能有上千个 task，每个都可能打开一些文件。很明显，master 需要花很长时间去处理这些请求，会承受很大的压力。

MCKUSICK 现在，在当前的 GFS 模式里，一个 GFS cell 有一个 master，对吗？

QUINLAN 是的。

MCKUSICK 历史上，你们曾经是一个数据中心一个 GFS cell，是吗？

QUINLAN 那是最初的目标，但是那样无法进行更大的扩展—一部分是由于受单 master 设计的限制，一部分原因是因为在一个 cell 里很难做到隔离性。结果，最终每个数据中心都不止一个 cell。而且后来我们也采用了一个称为”multi-cell”的策略，主要是可以把多个 GFS master 建立在一堆的 chunkserver 机器池上。通过这种方式，chunkserver 机器集可以配置成多个 master，比如说安排给它们 8 个 GFS master。这样应用程序负责在不同的 cell 间划分数据。

MCKUSICK 这样看来，每个程序就可以有自己的 master 来管理它自己的小文件系统。是这样的吗？

QUINLAN 好吧，有对也有错。应用程序倾向于使用一个 master 或者一小集 master。我们也提供称为名字空间的东西，它是一种静态划分名字空间的方式，人们可以用它将这些事情与实际应用程序透明。以日志处理系统为例：一旦日志超过了一个 cell 的存储能力，它们就会移到多个 GFS cell；一个名字空间文件描述了日志如何在这些不同的 cell 间划分，主要让这些具体的划分与应用程序透明。但是，这都是完全静态的。

MCKUSICK 整体来看，性能如何？

QUINLAN 我们没有投入很多精力在优化 master 的性能上。在 Google，很少将精力投入到对某个特殊的二进制执行程序的优化上。通常来说，我们只是让一切可以工作后，然后关注可扩展性—只要通过扩展某些东西就可以让性能得到提高。因为在这个问题中，我们有一个中央瓶颈可能会影响到很多操作，但是我们认为通过增加一些额外的努力让 master 变得更轻量级是更值得的事情。在将规模从数千操作扩展到数万的级别，单 master 还未成为瓶颈。当然在二进制程序上进行更多的优化肯定比不去做优化，更能让 GFS 走的更长久一些。

■

■

可以说让 GFS 在很短的时间内投入到产品级的应用中，为它的成功做出了贡献，同时也加速了 Google 走向市场的速度，最终可能导致了公司的成功。一个三人的团队负责所有的事情，包括 GFS 核心，使系统可以在一年内为部署做好准备。

但是所有成功的系统都会碰到一个问题—规模和应用总是以超过任何人可以想象的速度进行扩展。在 Google，这种情况则更加明显。

尽管各大公司组织没有就文件系统统计信息进行过交换共享，可以说 GFS 是世界上运行中的最大的文件系统。因此，即使原始的 GFS 架构已经提供了数倍的扩展能力，但是 Google 还是很快就超过了它。

另外，GFS 所需要支持的应用程序数据也在急剧增长。在对原始 GFS 的架构者 Howard Gobioff 的一次采访中，他提到"我们最早的 GFS 版本用户主要是为爬虫和索引系统。当质量保证团队和研究团队开始大量使用 GFS 的时候我们迎来了第二个高峰—而且他们都是用 GFS 来存储大的数据，很快我们就有了 50 个用户，他们随时都需要技术支持"。

令人吃惊的是 Google 不仅构建了这样一个文件系统，而且所有的应用程序都运行在它之上。需要对 GFS 进行持续调整以更好的适应新的使用场景，同时应用程序本身的也在伴随着 GFS 的优势和缺点不断演进。"因为一切都是我们构建的，因此我们可以做任何我们想做的"，Gobioff 一针见血的指出。"我们可以将问题在应用程序与文件系统之间来回考量，最后决定在哪块进行调整"。

关于规模的问题，需要一些更实质性的调整。一种上层的解决策略是使用多个跨网络的 cells，它们在功能上相关但是是不同的文件系统。除了有助于解决这种扩展性问题，这对于很多来自分散的数据中心的操作也是一种有效的安排。

快速的增长也会对初始的 GFS 设计的另一个关键参数造成压力：选择 64MB 作为标准 chunk 大小。当然，这比普通的文件系统块大小要大很多，但是这是因为由 Google 的爬虫和索引系统生成的文件本身很大。但是伴随着应用程序的多样性，必须找到一种方式让系统可以高效地处理那些大量远小于 64MB 的文件(比如 Gmail 中的文件)。文件数本身不是太大的问题，但是所有的文件在中央的 master 节点上都有一个内存需求，因此这也将原始 GFS 设计的固有风险暴露出来。

■
■
MCKUSICK 从最初的 GFS 论文上，可以看出文件数一直是一个关键的问题。能否深入讲一下？

QUINLAN 文件数问题很早就碰到了。举一个具体的例子，在我在 Google 的早期，设计过一个日志处理系统。最初设计的模型里，前端服务器会写 log，我们之后为处理和归档的需要，简单地将它拷贝到 GFS 上。一开始工作的很好，但是随着前端服务器的增加，每个每天都在产生日志。同时，日志的类型也在不断增加，还有前端服务器可能会陷入 crash 循环，这会产生更多的日志。这使得我们面对的日志规模远远超出了我们最初的估计。这成为我们必须要关注的问题。最后，我们不得不承认没有办法去应对这样的文件数增长规模。

MCKUSICK 不知道我理解的是否正确：你们关于文件数增长带来的问题是因为你们需要在 master 端为每个文件保存一些元数据，而这些元数据必须能够放入 master 的内存。

QUINLAN 对，是这样的。

MCKUSICK 那么这就只能容纳不能让 master 内存耗尽的有限数目的文件了？

QUINLAN 对。有两种元数据。一个用于标识文件，另一个用于组成文件的那些 chunks。如果你有一个只有 1MB 的 chunk，虽然它只占了 1MB 的磁盘空间，但是它仍然需要这两类放在 master 上的元数据。如果你的平均文件大小小于 64MB，那么你能存储下的对象数就会降低。这就成了问题。

回到前面的日志系统的例子，情况很快就很清楚了，我们考虑的这种自然性的映射根本是行不通的。我们需要找到一种方式来绕过这个问题，通过将一些底层对象合并成大文件的方式。在日志系统的这个例子里，虽然它不像造一个火箭那样复杂，但是还是需要付出一些努力。

MCKUSICK 这听起来有些像在旧时代里，IBM 因为具有磁盘分配上的限制，因此就提供给用户一个工具，可以将一系列文件放在一块，然后为它生成一个内容表格。

QUINLAN 是的。对于我们来说，每个应用程序都需要或多或少的这样去做。在

某些应用程序里这可能很简单。对于其他的一些应用程序，文件数问题可能更急切。大多数情况下，对于这种应用程序最简单的解决方案就是不要使用 GFS—即使最初看来文件数规模是可以接受的，但是它很快会成为一个问题。在我们开始使用更多的共享 cells 的时候，我们开始在文件数和存储空间上设置 quota。目前为止，人们遇到的大部分限制都是因为文件数 quota。与之相比，底层的存储 quota 很少成为一个问题。

MCKUSICK 为了解决文件数问题，你们采取什么样的长期策略？很明显，如果 master 仍然需要把所有的元数据存放在内存，即使是采用分布式的 master 也无法解决这个问题。

QUINLAN 分布式的 master 肯定会允许增加文件数，增加的数量与你想投入的机器数一致。这肯定是有帮助的。分布式多 master 模型有一个问题，如果你将所有的东西扩展两个数量级，然后文件平均大小降低到 1-MB，这与 64-MB 的情况仍然是有很大的不同。在 1MB 的情况下，你会遇到很多需要关注的其他问题。比如如果你要读取 10000 个 10-KB 文件，比仅仅读 100 个 1MB 文件你需要进行更多的 seek 操作。

我的直觉是如果你的设计是面向平均 1MB 大小的文件，那么肯定需要提供比面向平均 64MB 大小的设计多得多的东西。

MCKUSICK 那么你们做了哪些事情使得 GFS 可以在 1MB 文件的情况下工作？

QUINLAN 我们并没有修改现有的 GFS 设计。我们的计划为 1MB 文件提供服务的分布式 master 系统将会是一个全新的设计。我们的目标是每个 master 可以处理 100million 级别的文件，可以有数百个 master。

MCKUSICK 那么，每个 master 上面肯定不会有所有的数据吧？

QUINLAN 正是如此。

■

■

随着 Goolge Bigtable 最近的出现，一个用于管理结构化数据的分布式存储系统，针对文件数问题的一个潜在解决方案—尽管可能不是最好的一个，但起码是可用的一个。

然而 Bigtable 的意义远远超过了文件数问题。尤其是，它是设计用于 PB 级别，跨越成千上万台机器，简化系统机器的添加以及不需要重配置就可以自动化利用这些资源。对于公司来说，使用集中电力，潜在的冗余，大规模部署商品化硬件带来的成本节省，这些都是非常明显的优势。

目前 Bigtable 已经用于很多 Google 应用程序。尽管它代表与过去的一种完全不同的系统，但是需要说明的是，Bigtable 建立在、运行在 GFS 之上。而且很多方面的设计与大多数的 GFS 设计思想是一致的。这样看来，它也是使得 GFS 在快速

和广泛的变化中继续保持活力的一种大的改进。
■
■
MCKUSICK 你们现在已经有了 Bigtable。在你的角度看是否会将它看做是一个应用程序呢？

QUINLAN 从 GFS 的角度看，它的确是一个应用程序。但是很明显，它更是一种基础架构。

MCKUSICK 不知这样的理解是否正确：Bigtable 其实是一种轻量级的关系型数据库。

QUINLAN 它并不是一个真正的关系数据库。我是说，我们没有提供 SQL，实际上它也不支持像 join 这样的操作。Bigtable 实际上是一个允许你维护大量 key-value 对及其 schema 的一个结构化的存储系统。

MCKUSICK 实际中 Bigtable 的客户端都有哪些呢？

QUINLAN Bigtable 的使用还在增长中，目前它已用于爬虫和索引系统，同时还被很多 client-facing 的应用程序所使用。事实上，现在已经有成堆的 Bigtable client。基本上，具有任何大量小数据项的 app 都倾向于使用 Bigtable，尤其是在具有相当结构化的数据时。

MCKUSICK 我想我这里真正想提的问题是：Bigtable 是否单纯为了给应用程序提供一种处理小文件问题的尝试而提出的呢，主要方法就是通过将一堆小的东西聚合在一块？

QUINLAN 这肯定可以算作 Bigtable 的一个应用场景，但是它实际上是为了解决更通用的一类问题。如果你以那种方式使用 Bigtable—即，作为一种解决文件数问题的方式—那么你肯定不会充分利用 Bigtable 提供的功能。Bigtable 实际上并不是解决这种问题的一个理想方案，因为它可能给你的操作引入很多额外资源开销。而且，它的垃圾回收策略也不是侵入性的，这就无法最有效的利用你的空间。我想说的是，那些使用 Bigtable 单纯来解决文件数问题的人们可能并不会感到高兴，但是毫无疑问这是解决这个问题的一个方式。

MCKUSICK 根据我的了解，看起来这个想法只有两种基本的数据结构：logs 和 SSTables(sorted string tables)。因此，我猜 SSTables 肯定是用来处理 key-value 对及排序的，这与 Bigtable 有何区别？

QUINLAN 主要的区别是 SSTables 是不可变的，而 Bigtable 提供了可变的 key value 存储。Bigtable 本身实际上是建立在 logs 和 SSTables 之上。初始时，它会将输入数据存入一个事务日志文件，之后它会被 compacted 成一系列的 SSTables，随着时间的进行 SSTables 也会被 compacted。这让我想起了 log-structure 文件系统。不管怎样，如你所看到的，logs 和 SSTables 确实是我们对我们大部分数据进行结

构化的底层数据结构。我们使用 log 文件记录变更操作。一旦到达一定数量，就可以对它们排序，把它们放入一个具有索引的结构里。

尽管 GFS 并没有提供一个 Posix 接口，它仍然有一个漂亮的通用文件系统接口，因此人们可以自由的存储他们喜欢的任意数据。只是，随着时间的推移，我们的大多数用户最后只需要使用这样两种数据结构。我们也有一个称为 protocol buffer 的东西，是我们的数据描述语言。在两种数据结构中的大部分的数据都会是 protocol buffer 的格式。

同时它也提供了压缩和 checksums。即使有些人可能想重新实现这些东西，大部分的人只是直接使用这两个基本构建块有可以了。
■
■
因为 GFS 最初是为爬虫和索引系统设计的，吞吐率意味着一切。实际上，原始论文也明确指出了这一点。

但是 Google 也开发了很多面向用户的服务，对于它们来说，大部分都不符合上述情况。这使得 GFS 的单 master 设计的缺点更加迫切。一个单点失败对于面向批处理的应用来说可能不是一个灾难，但是对于延时敏感的应用来说肯定是不可忍受的，比如视频服务。即使后来增加了自动故障恢复的能力，但是服务仍可能长达 1 分钟的时间不可用。

当然这个对于 GFS 的挑战已经通过将延时敏感的应用程序建立在另一个设计在完全不同的优先级集合的文件系统上得到了解决。
■
■
MCKUSICK 文档里说的很明确，GFS 设计的最初的重点在批处理效率而不是低延迟。但是现在它已经引起了一些问题，尤其是在处理视频服务这样的东西时。你们是怎么解决这个问题的呢？

QUINLAN GFS 设计模型起初只是关注吞吐率，并没有关注应该达到怎样的延迟。举一个具体的例子，比如你准备写文件，它将会被写成一式三份的形式—意味着你实际上需要写到三个 chunkserver 上。如果其中一个死了或者很长时间内一直不稳定，GFS master 会发现这个问题，并引发一个称为 pullchunk 的过程，它会再复制一份 chunk 出来。使得你能达到三个 copy，之后系统才会将控制权交给客户端，然后继续写。

当我们执行一个 pullchunk 时，可能会限制在 5-10MB/s 的速度上。这样对于 64MB，就可能需要花费 10 秒钟。还有很多其他的情况会花掉 10 秒钟到一分钟，这对于批处理类型没有问题。比如你正在进行一个 MapReduce 任务，你会觉得对于一个几小时的任务来说，几分钟没什么影响。但是，如果你正在使用 Gmail，然后你尝试写入一个代表用户行为的变更，然后被卡住了一分钟，这的确会很糟糕。

起初，GFS 没有提供自动的故障恢复。都是一个手动的过程。尽管这不经常发生，

但是一旦发生，这个 GFS cell 就可能 down 掉一个小时。尽管我们最初的 master 故障恢复可能需要分钟级的时间，经过这些年后，现在大概需要数十秒的级别。

MCKUSICK 然而，对于面向用户的应用来说，这仍然是不可接受的。

QUINLAN 是的。当我们提供了故障恢复和错误恢复后，对于批处理的情况可能可接受的了，但是站在面向用户的应用程序的角度看，这些仍然是不行的。这里存在的另一个问题是，我们为提高吞吐率所进行的优化，将数千个操作放到队列中一块处理。这提高了吞吐率，但是却不利于延迟。光在等待队列中的等待时间就可能达到数秒钟。

现在，我们的使用场景已经从基于 MapReduce 的世界更多的转移到依赖于诸如 Bigtable 这样的交互式世界中。Gmail 是一个很明显的例子。视频的情况可能并不是那么糟糕，因为当你对数据进行流式传输的时候，意味着你可以进行缓冲。但是将一个交互式数据库建立在一个起初用于面向批处理操作的文件系统上，本身是一件很痛苦的事情。

MCKUSICK 你们又是怎样处理这些情况的呢？

QUINLAN 在 GFS 内部，我们会进行一定程度的改进，但主要是通过设计应用程序来处理遇到的问题。以 Bigtable 为例，Bigtable 的事务日志实际上是将事务日志化的最大瓶颈，我们通过打开两个文件进行日志写入，之后再归并它们来解决这个问题。我们倾向于设计具有类似功能的应用程序—让它们自己来隐藏延迟问题，因为系统底层并不擅长处理这种问题。

Gmail 的开发者们使用了一个多宿主模型，当你的 Gamil 账号所做的其中一个实例挂掉了，可以简单地通过转到另一个数据中心解决这个问题。实际上，这主要是用来保证可用性，其中一部分的原因也是为了隐藏 GFS 的问题。

MCKUSICK 我觉得，通过使用一个分布式 master 文件系统，一定可以解决其中某些延迟问题。

QUINLAN 这的确是我们的一个设计目标。而且，Bigtable 本身是一个失败感知迅速的系统，能够对失败迅速作出反应。使用它作为我们的元数据存储系统可以帮助解决一些延时问题。

■

■

GFS 做出了很多与原始文件系统不同的设计选择。对于一致性所采用的策略是其中非常明显的一个。GFS 的工程师团队选择了一个与传统文件系统相比，相对松的一致性保证。因为 GFS 主要是作为一个 append-only 系统使用，而不是一个 overwriting 系统。

■

■

MCKUSICK 我们来讨论下一致性吧。问题看起来好像是，要让所有的东西全部写入到所有副本想必要花大量的时间。我想在继续之前，你先说一下关于 GFS 必须要确保所有的都已经全部被写入后才能继续进行，这一点所带来的影响吧。

QUINLAN 是的。

MCKUSICK 如果这样的话，你们是怎么处理不一致的情况呢？

QUINLAN 客户端失败可能会把事情搞地很糟糕。基本上，GFS 的模型里，客户端仅仅是不断的推送写操作直到它成功为止。如果客户端在操作中 crash 掉，就会留下一些不确定的状态。

最初，这被认为是没有问题的，但是随着时间的推移，我们不断的压缩不一致性可以被容忍的窗口，然后不断的去降低这种不一致。只要数据处于不一致的状态，你都可能得到同一个文件的不同大小。这会导致一些问题。我们不得不在这些情况下提供一些检查文件数据一致性的接口。我们还有一个称为 RecordAppend 的东西，是设计用来允许多个 writer 并发向一个文件 append 的。在这里，一致性设计的很松。现在看来，这带来了比我们想象的多的痛苦。

MCKUSICK 为什么是 loose 的呢？如果主副本为每次写操作选择好在哪个 offset 开始写，然后保证这一定会发生，不太清楚不一致性是如何产生的。

QUINLAN 当主副本重试时就会产生。它会选择一个 offset，也会执行写操作，但是其中的某个写操作可能不会被实际写入。这样这个主副本就可能已经变化了，此时它可能会选出一个不同的 offset。RecordAppend 也不提供任何 replay 保护。在一个文件里你可能得到某个数据多次。

甚至还有些情况，你可能会以不同的顺序得到数据。比如数据可能在某个 chunk 副本中重复出现了多次，但是其他的副本里可能没有。如果你正在读取文件，那么读取多次就可能以多种方式得到数据。在记录级别上，你读到的记录顺序依赖于你刚好读取到哪个 chunk 副本。

MCKUSICK 这是 by design 的吗？

QUINLAN 当时看起来这是个好主意。但是现在看来，我认为这种一致性带来的痛苦也比它带来的好处要多。这与人们关于文件系统的期望不同，通常会让人感到很吃惊。

MCKUSICK 现在看的话，怎么处理这种不同？

QUINLAN 我认为让一个文件只有一个写者会更有意义。

MCKUSICK 恩，但是当有多个人想 append 某个日志时该怎么办呢？

QUINLAN 可以用一个进程将这些写操作进行串行化，来保证各副本的一致性。

MCKUSICK 而且当你想对一个 chunk 进行 snapshot 的时候也会有这个问题。另外，还有一些情况比如你有时想替换一个副本，或者当 chunkserver down 掉的时候，需要替换它的文件。

QUINLAN 事实上，这里有两种情况。第一个，如你所说，是恢复机制，肯定会引入关于文件副本的拷贝。在 GFS 里使用的方式是，我们撤销它上面的锁这样客户端就不能再去写它，当然这会引起一些延迟方面的问题。

还有另一个问题，是为了支持 GFS 的 snapshot。GFS 具有你所能想象的最通用的 snapshot 能力。你可以对任何目录进行 snapshot，同时与生成的拷贝是完全等价的。它们会共享未更新的数据。因此与大多数人关于 snapshot 的想法相比，它更多的是 clone 的概念。它很有趣，但是也带来了一些困难—尤其是当你试图创建更多的分布式系统以及文件树的更大的 chunks 进行 snapshot 时。

我觉得有趣的是 snapshot 也很少被用到尽管它是一个很强大的 feature。从文件系统的角度看，它实际上提供了一种很漂亮的功能，但是将 snapshot 引入文件系统，我想你也知道，实际上是很痛苦的。

MCKUSICK 我知道，我也这样做过。的确是让人难以忍受的—尤其是在一个 overwriting 系统中。

QUINLAN 是啊。无法否认的是，以实现的角度看，很难去创建真正的 snapshot。然而，看起来在这种情况下，尽力而为是一个正确的决定。同样地，它也是一个与我们早期所做的其他决定有趣的对比。

■

■

不管怎样，即使是在近十年后，这个关于 GFS 的报告看起来依然是很有意义的。虽然存在一些问题和缺点，但是毫无疑问的是 GFS 在 Google 的成功中起到了重要作用。但是，GFS 目前也面临着很多挑战。比如，在一个初始设计于批处理系统吞吐率的系统基础之上，去支持日益增长的面向用户的延时敏感的应用，其中的尴尬和问题也是很明显的。Bigtable 的出现对此提供了一些帮助。然而，实际上 Bigtable 也并没有解决 DFS 的所有问题。它只是减轻了系统的单 master 设计的瓶颈限制。因为这样那样的原因，Google 的工程师在过去的两年中一直在为实现一个新的分布式文件系统而努力，通过它来充分利用 Bigtable 的优势，去解决那些对于 GFS 来说很难解决的问题。

现在看来，为了保证 GFS 持续提供服务，在未来的时间里它将会依旧持续地演化。

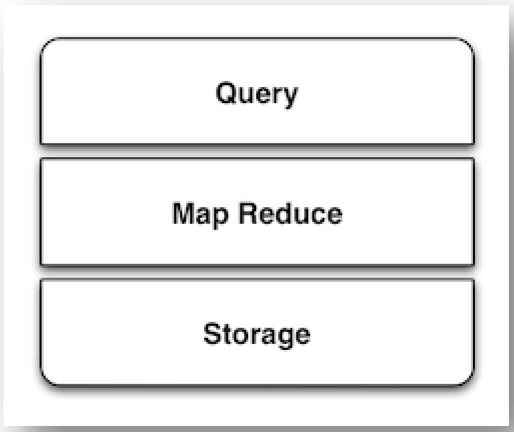
海量数据的存储计算和查询模型(译)

作者：Edd Dumbill 2010-09-22
译者：phylips@bmy 2011-01-16
出处：http://duanple.blog.163.com/blog/static/709717672011016103028473/

[说明：本文翻译自 The SMAQ stack for big data，SMAQ代表了存储(Storage)，计算(MapReduce)和(And)查询(Query)。]

海量数据(“Big Data”)是指那些足够大的数据，以至于无法再使用传统的方法进行处理。在过去，一直是 Web 搜索引擎的创建者们首当其冲的面对这个问题。而今天，各种社交网络、移动应用以及各种传感器和科学领域每天创建着上 PB 的数据。为了应对这种大规模数据处理的挑战，google 创造了 MapReduce。Google 的工作以及 yahoo 创建的 Hadoop 孵化出一个完整的海量数据处理工具的生态系统。

随着 MapReduce 的流行，一个由数据存储层，MapReduce 和查询(简称 SMAQ)组成的海量数据处理的堆栈模型也逐渐展现出来。SMAQ 系统通常是开源的、分布式的、运行在普通硬件上。

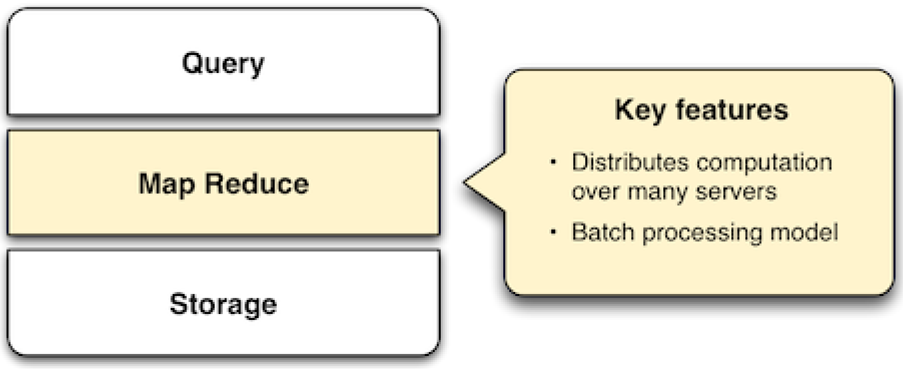


就像由 Linux, Apache, MySQL and PHP 组成的 LAMP 改变了互联网应用开发领域一样，SMAQ 将会把海量数据处理带入一个更广阔的天地。正如 LAMP 成为 Web2.0 的关键推动者一样，SMAQ 系统将支撑起一个创新的以数据为驱动的产品和服务的新时代。

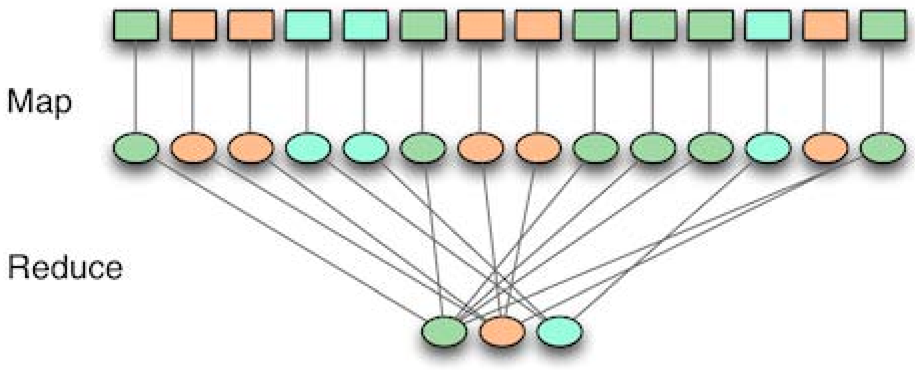
尽管基于 Hadoop 的架构占据了主导地位，但是 SMAQ 模型也包含大量的其他系统，包括主流的 NoSQL 数据库。这篇文章描述了 SMAQ 堆栈模型以及今天那些可以包括在这个框架下的海量数据处理工具。

MapReduce

MapReduce 是 google 为创建 web 网页索引而构建的。MapReduce 框架已成为今天大多数海量数据处理的厂房。MapReduce 的关键在于，将在数据集合上的一个查询进行划分，然后在多个节点上并行执行。这种分布式模式解决了数据太大以至于无法存放在单独一台机器上的难题。



为了理解 MapReduce 是如何工作的，我们首先看它名字所体现出的两个过程。首先在 map 阶段，输入数据被一项一项的处理，转换成一个中间结果集，然后在 reduce 阶段，这些中间结果又被规约产生一个我们所期望得到的归纳结果。



说到 MapReduce，通常要举的一个例子就是查找一篇文档中不同单词的出现个数。在 map 阶段单词被抽出来，然后给个 count 值 1，在 reduce 节点，将相同的单词的 count 值累加起来。

看起来是不是将一个很简单的工作搞地很复杂了，这就是 MapReduce。为了让 MapReduce 完成这项任务，map 和 reduce 阶段必须遵守一定的限制来使得工作可以并行化。将查询请求转换为一个或者多个 MapReduce 并不是一个直观的过程，为了解决这个问题，一些更高级的抽象被提出来，我们将在下面关于查询的那节里进行讨论。

使用 MapReduce 解决问题，通常需要三个操作：

- 数据加载—用数据仓库的叫法，这个过程叫做抽取(extract),转换(transform),加载(load)｛简称 ETL｝更合适些。为了利用 MapReduce 进行处理，数据必须从源数据里抽取出来，进行必要的结构化，加载到 MapReduce 可以访问的存储层。
- MapReduce—从存储层访问数据，进行处理，再将结果返回给存储层
- 结果抽取—一旦处理完毕，为了让结果对于人来说是可用的，还需要能够将存储层的结果数据进行查询和展示。

很多 SMAQ 系统都具有自身的一些属性，主要就是围绕上述三个过程的简化。

Hadoop MapReduce

Hadoop是主要的开源MapReduce实现。由yahoo资助，2006 年由 Doug Cutting创建，2008 年达到了web规模的数据处理容量。Hadoop项目现在由Apache管理。随着不断的努力，和多个子项目一起共同构成了完整的SMAQ模型。

由于是用 java 实现的，所以 Hadoop 的 MapReduce 实现可以通过 java 语言交互。创建 MapReduce job 通常需要写一些函数用来实现 map 和 reduce 阶段需要做的计算。处理数据必须能够加载到 Hadoop 的分布式文件系统中。

以 wordcount 为例，map 函数如下(来源于 Hadoop MapReduce 文档，展示了其中关键的步骤)

```
public static class Map
        extends Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
                String line = value.toString();
                StringTokenizer tokenizer = new
StringTokenizer(line);
                while (tokenizer.hasMoreTokens()) {
                        word.set(tokenizer.nextToken());
                        context.write(word, one);
                }
        }
}
```

对应的 reduce 函数如下：

```
public static class Reduce
                extends Reducer<Text, IntWritable, Text, IntWritable>
{
        public void reduce(Text key, Iterable<IntWritable> values,
                Context context) throws IOException,
InterruptedException {
                int sum = 0;
                for (IntWritable val : values) {
                        sum += val.get();
                }
                context.write(key, new IntWritable(sum));
        }
}
```

使用 Hadoop 运行一个 MapReduce job 包括如下几个步骤：

1. 用一个 java 程序定义 MapReduce 的各个阶段
2. 将数据加载进文件系统
3. 提交 job 进行执行
4. 从文件系统获取执行结果

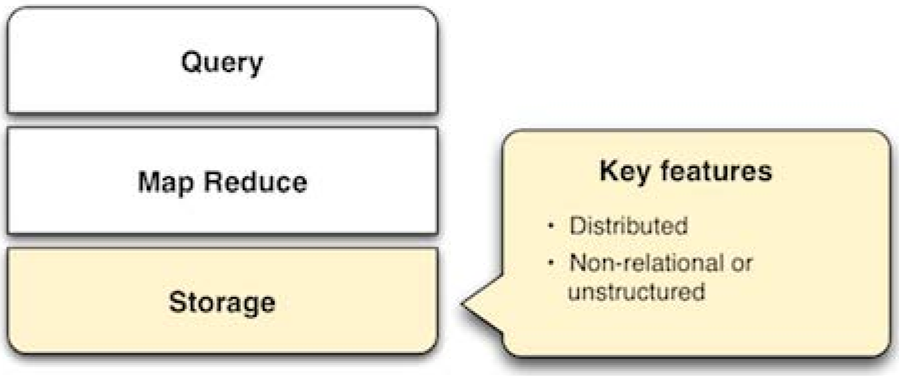
直接通过 java API，Hadoop MapReduce job 写起来可能很复杂，需要程序员很多方面的参与。为了让数据加载和处理工作更加简单直接，围绕着 Hadoop 一个很大的生态系统已经形成。

其他实现

MapReduce已经在很多其他的程序语言和系统中实现，详细的列表可以参考 Wikipedia's entry for MapReduce.。尤其是几个NoSQL数据库已经集成了MapReduce，后面我们会对此进行描述。

Storage

从数据获取到结果存放，MapReduce 都需要与存储打交道。与传统数据库不同，MapReduce 的输入数据并不是关系型的。输入数据存放在不同的 chunk 上，能够划分给不同的节点，然后提供以 key-value 的形式提供给 map 阶段。数据不需要一个 schema，而且可能是无结构的。但是数据必须是可分布的，能够提供给不同的处理节点。



存储层的设计和特点很重要不仅仅是因为它与 MapReduce 的接口，而且因为它们直接决定了数据加载、结果查询和展示的方便性。

Hadoop 分布式文件系统

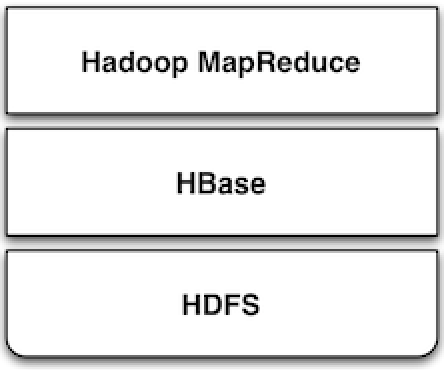
Hadoop使用的标准存储机制是HDFS。作为 Hadoop的核心部分，HDFS有如下特点，详细参见 HDFS design document.：

- 容错 – 假设失败是常态，允许 HDFS 运行在普通硬件上
- 流数据访问 – HDFS 实现时考虑的是批量处理，因此着重于高吞吐率而不是数据的随机访问
- 高度可扩展性 – HDFS 可以扩展到 PB 级的数据，比如 Facebook 就有一个这样的产品级使用
- 可移植性 – Hadoop 是可以跨操作系统移植的
- 单次写 – 假设文件写后不会改变，HDFS 简化了 replication 提高了数据吞吐率
- 计算本地化 – 考虑到数据量，通常将程序移到数据附近执行会更快，HDFS 提供了这方面的支持

HDFS 提供了一个类似于标准文件系统的接口。与传统数据库不同，HDFS 只能进行数据存储和访问，而不能为数据建立索引。无法对数据进行简单的随机访问。但是一些更高级的抽象已经创建出来，用来提供对 Hadoop 的更细粒度的功能，比如 HBase。

HBase,Hadoop 数据库

一种使 HDFS 更具可用性的方法是 HBase。模仿谷歌的 BigTable 数据库，HBase 也是一个设计用来存储海量数据的列存式数据库。它也属于 NoSQL 数据库范畴，类似于 Cassandra 和 Hypertable。



HBase 使用 HDFS 作为底层存储系统，因此也具有通过大量容错分布式节点来存储大量的数据的能力。与其他的列存储数据库类似，HBase 也提供基于 REST 和 Thrift 的访问 API。

由于创建了索引，HBase 可以为一些简单的查询提供对内容快速的随机访问。对于复杂的操作，HBase 为 Hadoop MapReduce 提供数据源和存储目标。因此 HBase 允许系统以数据库的方式与 MapReduce 进行交互，而不是通过底层的 HDFS。

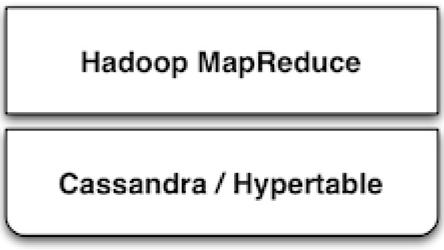
Hive

数据仓库，或者是提供使数据报告和分析更简单的存储方式是 SMAQ 系统的一个重要应用领域。最初在 Facebook 开发的 Hive，是一个建立在 Hadoop 之上的数据仓库框架。类似于 HBase，Hive 提供一个在 HDFS 上的基于表的抽象，简化了结构化数据的加载。与 HBase 相比，Hive 只能运行 MapReduce job 进行批量数据分析。如下面查询那部分描述的，Hive 提供了一个类 SQL 的查询语言来执行 MapReduce job。

Cassandra and Hypertable

Cassandra 和 Hypertable 都是具有 BigTable 模式的类似于 HBase 的列存储数据库。

作为 Apache 的一个项目，Cassandra 最初是在 Facebook 产生的。现在应用在很多大规模的 web 站点，包括 Twitter, Facebook, Reddit and Digg。Hypertable 产生于 Zvents，现在也是一个开源项目。



这两个数据库都提供与 Hadoop MapReduce 交互的接口，允许它们作为 Hadoop MapReduce job 的数据源和目标。在更高层次上，Cassandra 提供与 Pig 查询语言的集成(参见查询章节)，而 Hypertable 已经与 Hive 集成。

NoSQL 数据库的 MapReduce 实现

目前为止我们提到的存储解决方案都是依赖于 Hadoop 进行 MapReduce。还有一些 NoSQL 数据库为了对存储数据进行并行计算本身具有内建的 Mapreduce 支持。与 Hadoop 系统的多组件 SMAQ 架构不同，它们提供一个由 Storage, MapReduce and Query 一体组成的自包含系统。

基于 Hadoop 的系统通常是面向批量处理分析，NoSQL 存储通常是面向实时应用。在这些数据库里，MapReduce 通常只是一个附加功能，作为其他查询机制的一个补充而存在。比如，在 Riak 里 对 MapReduce job 通常有一个 60 秒的超时限制，而通常来说， Hadoop 认为一个 job 可能运行数分钟或者数小时。

下面的这些 NoSQL 数据库都具有 MapReduce 功能：

- CouchDB，一个分布式数据库，提供了半结构化的文档存储功能。主要特点是提供很强的多副本支持，以及可以进行分布式更新。在 CouchDB 里，查询是通过使用 javascript 定义 MapReduce 的 map 和 reduce 阶段实现的。
- MongoDB，本身很类似于 CouchDB，但是更注重性能，对于分布式更新，副本，版本的支持相对弱些。MapReduce 也是通过 javascript 描述的。
- Riak，与前面两个数据库也很类似。但是更关注高可用性。可以使用 javascript 或者 Erlang 描述 MapReduce。

与关系型数据库的集成

在很多应用中，主要的源数据存储在关系型数据库中，比如 Mysql 或者 Oracle。MapReduce 通常通过两种方式使用这些数据：

- 使用关系型数据库作为源(比如社交网络中的朋友列表)
- 将 MapReduce 结果重新注入到关系型数据库(比如基于朋友的兴趣产生的产品推荐列表)

理解 MapReduce 如何与关系型数据库交互是很重要的。最简单的，通过组合使用 SQL 导出命令和 HDFS 操作，带分隔符的文本文件可以作为传统关系型数据库

和 Hadoop 系统间的导入导出格式。更进一步的讲，还存在一些更复杂的工具。

Sqoop 工具是设计用来将数据从关系型数据库导入到 Hadoop 系统。它是由 Cloudera(一个专注于企业级应用的 Hadoop 平台服务提供商)开发的。Sqoop 是与具体数据库无关的，因为它使用了 java 的 JDBC 数据库 API。可以将整个表导入，也可以使用查询命令限制需要导入的数据。

Sqoop 也提供将 MapReduce 的结果从 HDFS 导回关系型数据库的功能。因为 HDFS 是一个文件系统，所以 Sqoop 需要以分隔符标识的文本为输入，需要将它们转换为相应的 SQL 命令才能将数据插入到数据库。

对于 Hadoop 系统来说，通过使用 Cascading API 中的 cascading.jdbc 和 cascading-dbmigrate 也能实现类似的功能。

与 streaming 数据源的集成

关系型数据库以及流式数据源(比如 web 服务器日志，传感器输出)组成了海量数据系统的最常见的数据来源。Cloudera 的 Flume 项目就是旨在提供流式数据源与 Hadoop 之间集成的方便工具。Flume 收集来自于集群机器上的数据，将它们不断的注入到 HDFS 中。Facebook 的 Scribe 服务器也提供类似的功能。

商业性的 SMAQ 解决方案

一些 MPP(massively parallel processing)数据库具有内建的 MapReduce 功能支持。MPP 数据库具有一个由并行运行的独立节点组成的分布式架构。它们的主要功能是数据仓库和分析，可以使用 SQL。

Greenplum：基于开源的 PostreSQL DBMS，运行在分布式硬件组成的集群上。MapReduce 作为 SQL 的补充，可以进行在 Greenplum 上的更快速更大规模的数据分析，减少了几个数量级的查询时间。Greenplum MapReduce 允许使用由数据库存储和外部数据源组成的混合数据。MapReduce 操作可以使用 Perl 或者 Python 函数进行描述。

Aster Data 的 nCluster 数据仓库系统也提供 MapReduce 支持。MapReduce 操作可以通过使用 Aster Data 的 SQL-MapReduce 技术调用。SQL-MapReduce 技术可以使 SQL 查询和通过各种语言(C#, C++, Java, R or Python)的源代码定义的 MapReduce job 组合在一块。

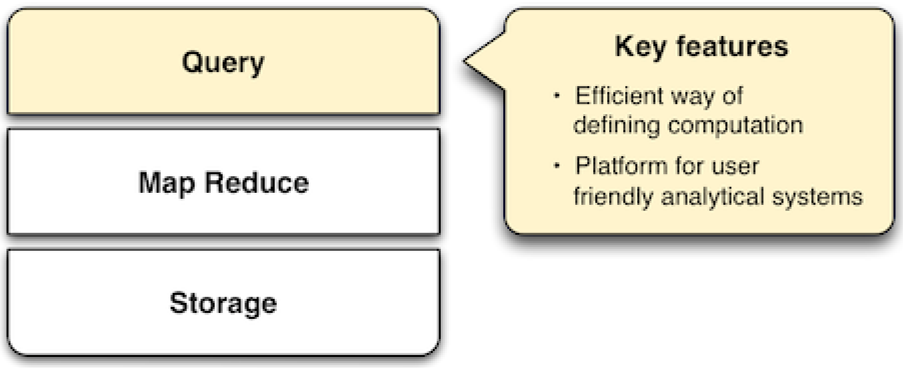
其他的一些数据仓库解决方案选择提供与 Hadoop 的连接器，而不是在内部集成 MapReduce 功能。
Vertica：是一个提供了 Hadoop 连接器的列存式数据库。
Netezza：最近由 IBM 收购。与 Cloudera 合作提高了它与 Hadoop 之间的互操作性。尽管它解决了类似的问题，但是实际上它已经不在我们的 SMAQ 模型定义之

内，因为它既不开源也不运行在普通硬件上。

尽管可以全部使用开源软件来创建一个基于 Hadoop 的系统，但是集成这样的一个系统仍然需要一些努力。Cloudera 的目的就是使得 Hadoop 更能适用于企业化的应用，而且在它们的 Cloudera Distribution for Hadoop (CDH)中已经提供一个统一的 Hadoop 发行版。

查询

通过上面的 java 代码可以看出使用程序语言定义 MapReduce job 的 map 和 reduce 过程并不是那么的直观和方便。为了解决这个问题，SMAQ 系统引人了一个更高层的查询层来简化 MapReduce 操作和结果查询。


Query
Map Reduce
Storage
Key features
Efficient way of defining computation
Platform for user friendly analytical systems

很多使用 Hadoop 的组织为了使操作更加方便，已经对 Hadoop 的 API 进行了内部的封装。有些已经成为开源项目或者商业性产品。
查询层通常并不仅仅提供用于描述计算过程的特性，而且支持对数据的存取以及简化在 MapReduce 集群上的执行流程。

Pig

由 yahoo 开发，目前是 Hadoop 项目的一部分。Pig 提供了一个称为 Pig Latin 的高级查询语言来描述和运行 MapReduce job。它的目的是让 Hadoop 更容易被那些熟悉 SQL 的开发人员访问，除了一个 Java API，它还提供一个交互式的接口。Pig 目前已经集成在 Cassandra 和 HBase 数据库中。

下面是使用 Pig 写的上面的 wordcount 的例子，包括了数据的加载和存储过程($0 代表记录的第一个字段)。

```
input = LOAD 'input/sentences.txt' USING TextLoader();

words = FOREACH input GENERATE FLATTEN(TOKENIZE($0));

grouped = GROUP words BY $0;
```

```
counts = FOREACH grouped GENERATE group, COUNT(words);

ordered = ORDER counts BY $0;

STORE ordered INTO 'output/wordCount' USING PigStorage();
```

Pig是非常具有表达力的，它允许开发者通过UDFs(User Defined Functions)书写一些定制化的功能。这些UDF使用java语言书写。尽管它比MapReduce API更容易理解和使用，但是它要求用户去学习一门新的语言。某些程度上它与SQL有些类似，但是它又与SQL具有很大的不同，因此那些熟悉SQL的人们很难将它们的知识在这里重用。

Hive

正如前面所述，Hive 是一个建立在 Hadoop 之上的开源数据仓库。由 Facebook 创建，它提供了一个非常类似于 SQL 的查询语言，而且提供一个支持简单内建查询的 web 接口。因此它很适合于那些熟悉 SQL 的非开发者用户。

与 Pig 和 Cascading 都需要进行编译相比，Hive 的一个长处是提供即席查询。对于那些已经成熟的商务智能系统来说，Hive 是一个更自然的起点，因为它提供了一个对于非技术用户更加友好的接口。Cloudera 的 Hadoop 发行版里集成了 Hive，而且通过 HUE 项目提供了一个更高级的用户接口，使得用户可以提交查询并且监控 MapReduce job 的执行。

Cascading, the API Approach

Cascading 提供了一个对 Hadoop 的 MapReduce API 的包装以使它更容易被 java 应用程序使用。它只是一个为了让 MapReduce 集成到更大的系统中时更简单的一个包装层。Cascading 包括如下几个特性：

- 旨在简化 MapReduce job 定义的数据处理 API
- 一个控制 MapReduce job 在 Hadoop 集群上运行的 API
- 访问基于 Jvm 的脚本语言，比如 Jython, Groovy, or JRuby.
- 与 HDFS 之外的数据源的集成，包括 Amazon S3，web 服务器
- 提供 MapReduce 过程测试的验证机制

Cascading 的关键特性是它允许开发者将 MapReduce job 以流的形式进行组装，通过将选定的一些 pipes 连接起来。因此很适用于将 Hadoop 集成到一个更大的系统中。

Cascading 本身并不提供高级查询语言，由它而衍生出的一个叫 Cascalog 的开源项目完成了这项工作。Cascalog 通过使用 Clojure JVM 语言实现了一个类似于 Datalog 的查询语言。尽管很强大，Cascalog 仍然只是一个小范围内使用的语言，因为它既不像 Hive 那样提供一个类 SQL 的语言，也不像 Pig 那样是过程性的。下面是使用 Cascalog 完成的 wordcout 的例子：

```
      (defmapcatop split [sentence]
            (seq (.split sentence "\\s+")))
      (?<- (stdout) [?word ?count]
            (sentence ?s) (split ?s :> ?word)
            (c/count ?count))
```

使用 Solr 进行搜索

大规模数据系统的一个重要组件就是数据查询和摘要。数据库层比如 HBase 提供了对数据的简单访问，但是并不具备复杂的搜索能力。为了解决搜索问题。开源的搜索和索引平台 solr 通常与 NoSQL 组合使用。Solr 使用 Luence 搜索技术提供一个自包含的搜索服务器产品。比如，考虑一个社交网络数据库，MapReduce 可以使用一些合理的参数用来计算个人的影响力。这个分数值会被写回到数据库。使用 Solr 进行索引，允许在这个社交网络上进行一些操作，比如找到最有影响力的人。

最初在 CENT 开发，现在作为 Apache 项目的 Solr，已经从一个单一的文本搜索引擎逐步演化，目前已支持导航和结果聚类。此外，Solr 还可以管理在分布式服务器上的海量数据。这使得它成为在海量数据上进行查询的理想解决方案，以及构建商业智能系统的重要组件。

总结

MapReduce 尤其是 Hadoop 实现提供在普通服务器上进行分布式计算的强有力的方式。再加上分布式存储以及用户友好的查询机制，它们形成的 SMAQ 架构使得海量数据处理通过小型团队甚至个人开发也能实现。

现在对数据进行深入的分析以及创建依赖于复杂计算的数据产品已经变得很廉价。其结果已经深远的影响了数据分析和数据仓库领域的格局，降低了该领域的进入门槛，培养了新一代的产品，服务和组织方式。这种趋势在Mike Loukides的"What is Data Science?"报告中有更深入的诠释。

Linux 的出现仅仅通过一台摆在桌面上的 linux 服务器带给那些创新的开发者们以力量。SMAQ 拥有相同的潜力来提高数据中心的效率，促进组织边缘的创新，开启廉价创建数据驱动业务的新时代。

Web Search for a Planet: The Google Cluster Architecture

Amenable to extensive parallelization, Google's Web search application lets different queries run on different processors and, by partitioning the overall index, also lets a single query use multiple processors. To handle this workload, Google's architecture features clusters of more than 15,000 commodity-class PCs with fault-tolerant software. This architecture achieves superior performance at a fraction of the cost of a system built from fewer, but more expensive, high-end servers.

Luiz André Barroso
Jeffrey Dean
Urs Hölzle
Google

Few Web services require as much computation per request as search engines. On average, a single query on Google reads hundreds of megabytes of data and consumes tens of billions of CPU cycles. Supporting a peak request stream of thousands of queries per second requires an infrastructure comparable in size to that of the largest supercomputer installations. Combining more than 15,000 commodity-class PCs with fault-tolerant software creates a solution that is more cost-effective than a comparable system built out of a smaller number of high-end servers.

Here we present the architecture of the Google cluster, and discuss the most important factors that influence its design: energy efficiency and price-performance ratio. Energy efficiency is key at our scale of operation, as power consumption and cooling issues become significant operational factors, taxing the limits of available data center power densities.

Our application affords easy parallelization: Different queries can run on different processors, and the overall index is partitioned so that a single query can use multiple processors. Consequently, peak processor performance is less important than its price/performance. As such, Google is an example of a throughput-oriented workload, and should benefit from processor architectures that offer more on-chip parallelism, such as simultaneous multithreading or on-chip multiprocessors.

Google architecture overview

Google's software architecture arises from two basic insights. First, we provide reliability in software rather than in server-class hardware, so we can use commodity PCs to build a high-end computing cluster at a low-end price. Second, we tailor the design for best aggregate request throughput, not peak server response time, since we can manage response times by parallelizing individual requests.

We believe that the best price/performance tradeoff for our applications comes from fashioning a reliable computing infrastructure from clusters of unreliable commodity PCs. We provide reliability in our environment at the software level, by replicating services across many different machines and automatically detecting and handling failures. This software-based reliability encompasses many different areas and involves all parts of our system design. Examining the control flow in handling a query provides insight into the high-level structure of the query-serving system, as well as insight into reliability considerations.

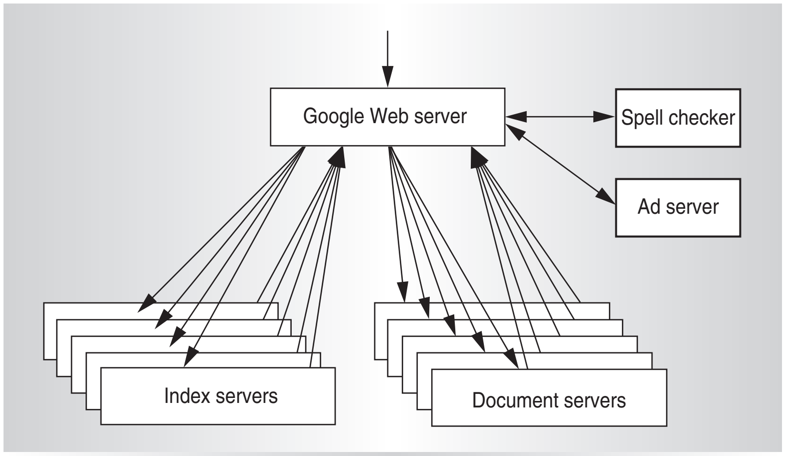


Figure 1. Google query-serving architecture.

Serving a Google query

When a user enters a query to Google (such as www.google.com/search?q=ieee+society), the user's browser first performs a domain name system (DNS) lookup to map www.google.com to a particular IP address. To provide sufficient capacity to handle query traffic, our service consists of multiple clusters distributed worldwide. Each cluster has around a few thousand machines, and the geographically distributed setup protects us against catastrophic data center failures (like those arising from earthquakes and large-scale power failures). A DNS-based load-balancing system selects a cluster by accounting for the user's geographic proximity to each physical cluster. The load-balancing system minimizes round-trip time for the user's request, while also considering the available capacity at the various clusters.

The user's browser then sends a hypertext transport protocol (HTTP) request to one of these clusters, and thereafter, the processing of that query is entirely local to that cluster. A hardware-based load balancer in each cluster monitors the available set of Google Web servers (GWSs) and performs local load balancing of requests across a set of them. After receiving a query, a GWS machine coordinates the query execution and formats the results into a Hypertext Markup Language (HTML) response to the user's browser. Figure 1 illustrates these steps.

Query execution consists of two major phases.[1] In the first phase, the index servers consult an inverted index that maps each query word to a matching list of documents (the hit list). The index servers then determine a set of relevant documents by intersecting the hit lists of the individual query words, and they compute a relevance score for each document. This relevance score determines the order of results on the output page.

The search process is challenging because of the large amount of data: The raw documents comprise several tens of terabytes of uncompressed data, and the inverted index resulting from this raw data is itself many terabytes of data. Fortunately, the search is highly parallelizable by dividing the index into pieces (*index shards*), each having a randomly chosen subset of documents from the full index. A pool of machines serves requests for each shard, and the overall index cluster contains one pool for each shard. Each request chooses a machine within a pool using an intermediate load balancer—in other words, each query goes to one machine (or a subset of machines) assigned to each shard. If a shard's replica goes down, the load balancer will avoid using it for queries, and other components of our cluster-management system will try to revive it or eventually replace it with another machine. During the downtime, the system capacity is reduced in proportion to the total fraction of capacity that this machine represented. However, service remains uninterrupted, and all parts of the index remain available.

The final result of this first phase of query execution is an ordered list of document identifiers (*docids*). As Figure 1 shows, the second phase involves taking this list of docids and computing the actual title and uniform resource locator of these documents, along with a query-specific document summary. Document servers (docservers) handle this job, fetching each document from disk to extract the title and the keyword-in-context snippet. As with the index lookup phase, the strategy is to partition the processing of all documents by

- randomly distributing documents into smaller shards
- having multiple server replicas responsible for handling each shard, and
- routing requests through a load balancer.

The docserver cluster must have access to an online, low-latency copy of the entire Web. In fact, because of the replication required for performance and availability, Google stores dozens of copies of the Web across its clusters.

In addition to the indexing and document-serving phases, a GWS also initiates several other ancillary tasks upon receiving a query, such as sending the query to a spell-checking system and to an ad-serving system to generate relevant advertisements (if any). When all phases are complete, a GWS generates the appropriate HTML for the output page and returns it to the user's browser.

Using replication for capacity and fault-tolerance

We have structured our system so that most accesses to the index and other data structures involved in answering a query are read-only: Updates are relatively infrequent, and we can often perform them safely by diverting queries away from a service replica during an update. This principle sidesteps many of the consistency issues that typically arise in using a general-purpose database.

We also aggressively exploit the very large amounts of inherent parallelism in the application: For example, we transform the lookup of matching documents in a large index into many lookups for matching documents in a set of smaller indices, followed by a relatively inexpensive merging step. Similarly, we divide the query stream into multiple streams, each handled by a cluster. Adding machines to each pool increases serving capacity, and adding shards accommodates index growth. By parallelizing the search over many machines, we reduce the average latency necessary to answer a query, dividing the total computation across more CPUs and disks. Because individual shards don't need to communicate with each other, the resulting speedup is nearly linear. In other words, the CPU speed of the individual index servers does not directly influence the search's overall performance, because we can increase the number of shards to accommodate slower CPUs, and vice versa. Consequently, our hardware selection process focuses on machines that offer an excellent request throughput for our application, rather than machines that offer the highest single-thread performance.

In summary, Google clusters follow three key design principles:

- *Software reliability.* We eschew fault-tolerant hardware features such as redundant power supplies, a redundant array of inexpensive disks (RAID), and high-quality components, instead focusing on tolerating failures in software.
- *Use replication for better request throughput and availability.* Because machines are inherently unreliable, we replicate each of our internal services across many machines. Because we already replicate services across multiple machines to obtain sufficient capacity, this type of fault tolerance almost comes for free.
- *Price/performance beats peak performance.* We purchase the CPU generation that currently gives the best performance per unit price, not the CPUs that give the best absolute performance.
- *Using commodity PCs reduces the cost of computation.* As a result, we can afford to use more computational resources per query, employ more expensive techniques in our ranking algorithm, or search a larger index of documents.

Leveraging commodity parts

Google's racks consist of 40 to 80 x86-based servers mounted on either side of a custom made rack (each side of the rack contains twenty 20u or forty 1u servers). Our focus on price/performance favors servers that resemble mid-range desktop PCs in terms of their components, except for the choice of large disk drives. Several CPU generations are in active service, ranging from single-processor 533-MHz Intel-Celeron-based servers to dual 1.4-GHz Intel Pentium III servers. Each server contains one or more integrated drive electronics (IDE) drives, each holding 80 Gbytes. Index servers typically have less disk space than document servers because the former have a more CPU-intensive workload. The servers on each side of a rack interconnect via a 100-Mbps Ethernet switch that has one or two gigabit uplinks to a core gigabit switch that connects all racks together.

Our ultimate selection criterion is cost per query, expressed as the sum of capital expense (with depreciation) and operating costs (hosting, system administration, and repairs) divided by performance. Realistically, a server will not last beyond two or three years, because of its disparity in performance when compared to newer machines. Machines older than three years are so much slower than current-generation machines that it is difficult to achieve proper load distribution and configuration in clusters containing both types. Given the relatively short amortization period, the equipment cost figures prominently in the overall cost equation.

Because Google servers are custom made, we'll use pricing information for comparable PC-based server racks for illustration. For example, in late 2002 a rack of 88 dual-CPU 2-GHz Intel Xeon servers with 2 Gbytes of RAM and an 80-Gbyte hard disk was offered on RackSaver.com for around $278,000. This figure translates into a monthly capital cost of $7,700 per rack over three years. Personnel and hosting costs are the remaining major contributors to overall cost.

The relative importance of equipment cost makes traditional server solutions less appealing for our problem because they increase performance but decrease the price/performance. For example, four-processor motherboards are expensive, and because our application parallelizes very well, such a motherboard doesn't recoup its additional cost with better performance. Similarly, although SCSI disks are faster and more reliable, they typically cost two or three times as much as an equal-capacity IDE drive.

The cost advantages of using inexpensive, PC-based clusters over high-end multiprocessor servers can be quite substantial, at least for a highly parallelizable application like ours. The example $278,000 rack contains 176 2-GHz Xeon CPUs, 176 Gbytes of RAM, and 7 Tbytes of disk space. In comparison, a typical x86-based server contains eight 2-GHz Xeon CPUs, 64 Gbytes of RAM, and 8 Tbytes of disk space; it costs about $758,000.[2] In other words, the multiprocessor server is about three times more expensive but has 22 times fewer CPUs, three times less RAM, and slightly more disk space. Much of the cost difference derives from the much higher interconnect bandwidth and reliability of a high-end server, but again, Google's highly redundant architecture does not rely on either of these attributes.

Operating thousands of mid-range PCs instead of a few high-end multiprocessor servers incurs significant system administration and repair costs. However, for a relatively homogenous application like Google, where most servers run one of very few applications, these costs are manageable. Assuming tools to install and upgrade software on groups of machines are available, the time and cost to maintain 1,000 servers isn't much more than the cost of maintaining 100 servers because all machines have identical configurations. Similarly, the cost of monitoring a cluster using a scalable application-monitoring system does not increase greatly with cluster size. Furthermore, we can keep repair costs reasonably low by batching repairs and ensuring that we can easily swap out components with the highest failure rates, such as disks and power supplies.

The power problem

Even without special, high-density packaging, power consumption and cooling issues can become challenging. A mid-range server with dual 1.4-GHz Pentium III processors draws about 90 W of DC power under load: roughly 55 W for the two CPUs, 10 W for a disk drive, and 25 W to power DRAM and the motherboard. With a typical efficiency of about 75 percent for an ATX power supply, this translates into 120 W of AC power per server, or roughly 10 kW per rack. A rack comfortably fits in 25 ft^2 of space, resulting in a power density of 400 W/ft^2. With higher-end processors, the power density of a rack can exceed 700 W/ft^2.

Table 1. Instruction-level measurements on the index server.

Characteristic	Value
Cycles per instruction	1.1
Ratios (percentage)	
Branch mispredict	5.0
Level 1 instruction miss*	0.4
Level 1 data miss*	0.7
Level 2 miss*	0.3
Instruction TLB miss*	0.04
Data TLB miss*	0.7

* Cache and TLB ratios are per instructions retired.

Unfortunately, the typical power density for commercial data centers lies between 70 and 150 W/ft^2, much lower than that required for PC clusters. As a result, even low-tech PC clusters using relatively straightforward packaging need special cooling or additional space to bring down power density to that which is tolerable in typical data centers. Thus, packing even more servers into a rack could be of limited practical use for large-scale deployment as long as such racks reside in standard data centers. This situation leads to the question of whether it is possible to reduce the power usage per server.

Reduced-power servers are attractive for large-scale clusters, but you must keep some caveats in mind. First, reduced power is desirable, but, for our application, it must come without a corresponding performance penalty: What counts is watts per unit of performance, not watts alone. Second, the lower-power server must not be considerably more expensive, because the cost of depreciation typically outweighs the cost of power. The earlier-mentioned 10 kW rack consumes about 10 MW-h of power per month (including cooling overhead). Even at a generous 15 cents per kilowatt-hour (half for the actual power, half to amortize uninterruptible power supply [UPS] and power distribution equipment), power and cooling cost only $1,500 per month. Such a cost is small in comparison to the depreciation cost of $7,700 per month. Thus, low-power servers must not be more expensive than regular servers to have an overall cost advantage in our setup.

Hardware-level application characteristics

Examining various architectural characteristics of our application helps illustrate which hardware platforms will provide the best price/performance for our query-serving system. We'll concentrate on the characteristics of the index server, the component of our infrastructure whose price/performance most heavily impacts overall price/performance. The main activity in the index server consists of decoding compressed information in the inverted index and finding matches against a set of documents that could satisfy a query. Table 1 shows some basic instruction-level measurements of the index server program running on a 1-GHz dual-processor Pentium III system.

The application has a moderately high CPI, considering that the Pentium III is capable of issuing three instructions per cycle. We expect such behavior, considering that the application traverses dynamic data structures and that control flow is data dependent, creating a significant number of difficult-to-predict branches. In fact, the same workload running on the newer Pentium 4 processor exhibits nearly twice the CPI and approximately the same branch prediction performance, even though the Pentium 4 can issue more instructions concurrently and has superior branch prediction logic. In essence, there isn't that much exploitable instruction-level parallelism (ILP) in the workload. Our measurements suggest that the level of aggressive out-of-order, speculative execution present in modern processors is already beyond the point of diminishing performance returns for such programs.

A more profitable way to exploit parallelism for applications such as the index server is to leverage the trivially parallelizable computation. Processing each query shares mostly read-only data with the rest of the system, and constitutes a work unit that requires little communication. We already take advantage of that at the cluster level by deploying large numbers of inexpensive nodes, rather than fewer high-end ones. Exploiting such abundant thread-level parallelism at the microarchitecture level appears equally promising. Both simultaneous multithreading (SMT) and chip multiprocessor (CMP) architectures target thread-level parallelism and should improve the performance of many of our servers. Some early experiments with a dual-context (SMT) Intel Xeon processor show more than a 30 percent performance improvement over a single-context setup. This speedup is at the upper bound of improvements reported by Intel for their SMT implementation.[3]

We believe that the potential for CMP systems is even greater. CMP designs, such as Hydra[4] and Piranha,[5] seem especially promising. In these designs, multiple (four to eight) simpler, in-order, short-pipeline cores replace a complex high-performance core. The penalties of in-order execution should be minor given how little ILP our application yields, and shorter pipelines would reduce or eliminate branch mispredict penalties. The available thread-level parallelism should allow near-linear speedup with the number of cores, and a shared L2 cache of reasonable size would speed up interprocessor communication.

Memory system

Table 1 also outlines the main memory system performance parameters. We observe good performance for the instruction cache and instruction translation look-aside buffer, a result of the relatively small inner-loop code size. Index data blocks have no temporal locality, due to the sheer size of the index data and the unpredictability in access patterns for the index's data block. However, accesses within an index data block do benefit from spatial locality, which hardware prefetching (or possibly larger cache lines) can exploit. The net effect is good overall cache hit ratios, even for relatively modest cache sizes.

Memory bandwidth does not appear to be a bottleneck. We estimate the memory bus utilization of a Pentium-class processor system to be well under 20 percent. This is mainly due to the amount of computation required (on average) for every cache line of index data brought into the processor caches, and to the data-dependent nature of the data fetch stream. In many ways, the index server's memory system behavior resembles the behavior reported for the Transaction Processing Performance Council's benchmark D (TPC-D).[6] For such workloads, a memory system with a relatively modest sized L2 cache, short L2 cache and memory latencies, and longer (perhaps 128 byte) cache lines is likely to be the most effective.

Large-scale multiprocessing

As mentioned earlier, our infrastructure consists of a massively large cluster of inexpensive desktop-class machines, as opposed to a smaller number of large-scale shared-memory machines. Large shared-memory machines are most useful when the computation-to-communication ratio is low; communication patterns or data partitioning are dynamic or hard to predict; or when total cost of ownership dwarfs hardware costs (due to management overhead and software licensing prices). In those situations they justify their high price tags.

At Google, none of these requirements apply, because we partition index data and computation to minimize communication and evenly balance the load across servers. We also produce all our software in-house, and minimize system management overhead through extensive automation and monitoring, which makes hardware costs a significant fraction of the total system operating expenses. Moreover, large-scale shared-memory machines still do not handle individual hardware component or software failures gracefully, with most fault types causing a full system crash. By deploying many small multiprocessors, we contain the effect of faults to smaller pieces of the system. Overall, a cluster solution fits the performance and availability requirements of our service at significantly lower costs.

At first sight, it might appear that there are few applications that share Google's characteristics, because there are few services that require many thousands of servers and petabytes of storage. However, many applications share the essential traits that allow for a PC-based cluster architecture. As long as an application orientation focuses on the price/performance and can run on servers that have no private state (so servers can be replicated), it might benefit from using a similar architecture. Common examples include high-volume Web servers or application servers that are computationally intensive but essentially stateless. All of these applications have plenty of request-level parallelism, a characteristic exploitable by running individual requests on separate servers. In fact, larger Web sites already commonly use such architectures.

At Google's scale, some limits of massive server parallelism do become apparent, such as the limited cooling capacity of commercial data centers and the less-than-optimal fit of current CPUs for throughput-oriented applications. Nevertheless, using inexpensive PCs to handle Google's large-scale computations has drastically increased the amount of computation we can afford to spend per query, thus helping to improve the Internet search experience of tens of millions of users. MICRO

Acknowledgments

Over the years, many others have made contributions to Google's hardware architecture that are at least as significant as ours. In particular, we acknowledge the work of Gerald Aigner, Ross Biro, Bogdan Cocosel, and Larry Page.

References

1. S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," *Proc. Seventh World Wide Web Conf.* (WWW7), International World Wide Web Conference Committee (IW3C2), 1998, pp. 107-117.
2. "TPC Benchmark C Full Disclosure Report for IBM eserver xSeries 440 using Microsoft SQL Server 2000 Enterprise Edition and Microsoft Windows .NET Datacenter Server 2003, TPC-C Version 5.0," http://www.tpc.org/results/FDR/TPCC/ibm.x4408way.c5.fdr.02110801.pdf.
3. D. Marr et al., "Hyper-Threading Technology Architecture and Microarchitecture: A Hypertext History," *Intel Technology J.*, vol. 6, issue 1, Feb. 2002.
4. L. Hammond, B. Nayfeh, and K. Olukotun, "A Single-Chip Multiprocessor," *Computer*, vol. 30, no. 9, Sept. 1997, pp. 79-85.
5. L.A. Barroso et al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," *Proc. 27th ACM Int'l Symp. Computer Architecture*, ACM Press, 2000, pp. 282-293.
6. L.A. Barroso, K. Gharachorloo, and E. Bugnion, "Memory System Characterization of Commercial Workloads," *Proc. 25th ACM Int'l Symp. Computer Architecture*, ACM Press, 1998, pp. 3-14.

Luiz André Barroso is a member of the Systems Lab at Google, where he has focused on improving the efficiency of Google's Web search and on Google's hardware architecture. Barroso has a BS and an MS in electrical engineering from Pontifícia Universidade Católica, Brazil, and a PhD in computer engineering from the University of Southern California. He is a member of the ACM.

Jeffrey Dean is a distinguished engineer in the Systems Lab at Google and has worked on the crawling, indexing, and query serving systems, with a focus on scalability and improving relevance. Dean has a BS in computer science and economics from the University of Minnesota and a PhD in computer science from the University of Washington. He is a member of the ACM.

Urs Hölzle is a Google Fellow and in his previous role as vice president of engineering was responsible for managing the development and operation of the Google search engine during its first two years. Hölzle has a diploma from the Eidgenössische Technische Hochschule Zürich and a PhD from Stanford University, both in computer science. He is a member of IEEE and the ACM.

Direct questions and comments about this article to Urs Hölzle, 2400 Bayshore Parkway, Mountain View, CA 94043; urs@google.com.

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google[*]

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both micro-benchmarks and real world use.

Categories and Subject Descriptors

D [**4**]: 3—*Distributed file systems*

General Terms

Design, reliability, performance, measurement

Keywords

Fault tolerance, scalability, data storage, clustered storage

[*]The authors can be reached at the following addresses: {*sanjay,hgobioff,shuntak*}*@google.com*.

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Second, files are huge by traditional standards. Multi-GB files are common. Each file typically contains many application objects such as web documents. When we are regularly working with fast growing data sets of many TBs comprising billions of objects, it is unwieldy to manage billions of approximately KB-sized files even when the file system could support it. As a result, design assumptions and parameters such as I/O operation and block sizes have to be revisited.

Third, most files are mutated by appending new data rather than overwriting existing data. Random writes within a file are practically non-existent. Once written, the files are only read, and often only sequentially. A variety of data share these characteristics. Some may constitute large repositories that data analysis programs scan through. Some may be data streams continuously generated by running applications. Some may be archival data. Some may be intermediate results produced on one machine and processed on another, whether simultaneously or later in time. Given this access pattern on huge files, appending becomes the focus of performance optimization and atomicity guarantees, while caching data blocks in the client loses its appeal.

Fourth, co-designing the applications and the file system API benefits the overall system by increasing our flexibility.

For example, we have relaxed GFS's consistency model to vastly simplify the file system without imposing an onerous burden on the applications. We have also introduced an atomic append operation so that multiple clients can append concurrently to a file without extra synchronization between them. These will be discussed in more details later in the paper.

Multiple GFS clusters are currently deployed for different purposes. The largest ones have over 1000 storage nodes, over 300 TB of disk storage, and are heavily accessed by hundreds of clients on distinct machines on a continuous basis.

2. DESIGN OVERVIEW

2.1 Assumptions

In designing a file system for our needs, we have been guided by assumptions that offer both challenges and opportunities. We alluded to some key observations earlier and now lay out our assumptions in more details.

- The system is built from many inexpensive commodity components that often fail. It must constantly monitor itself and detect, tolerate, and recover promptly from component failures on a routine basis.
- The system stores a modest number of large files. We expect a few million files, each typically 100 MB or larger in size. Multi-GB files are the common case and should be managed efficiently. Small files must be supported, but we need not optimize for them.
- The workloads primarily consist of two kinds of reads: large streaming reads and small random reads. In large streaming reads, individual operations typically read hundreds of KBs, more commonly 1 MB or more. Successive operations from the same client often read through a contiguous region of a file. A small random read typically reads a few KBs at some arbitrary offset. Performance-conscious applications often batch and sort their small reads to advance steadily through the file rather than go back and forth.
- The workloads also have many large, sequential writes that append data to files. Typical operation sizes are similar to those for reads. Once written, files are seldom modified again. Small writes at arbitrary positions in a file are supported but do not have to be efficient.
- The system must efficiently implement well-defined semantics for multiple clients that concurrently append to the same file. Our files are often used as producer-consumer queues or for many-way merging. Hundreds of producers, running one per machine, will concurrently append to a file. Atomicity with minimal synchronization overhead is essential. The file may be read later, or a consumer may be reading through the file simultaneously.
- High sustained bandwidth is more important than low latency. Most of our target applications place a premium on processing data in bulk at a high rate, while few have stringent response time requirements for an individual read or write.

2.2 Interface

GFS provides a familiar file system interface, though it does not implement a standard API such as POSIX. Files are organized hierarchically in directories and identified by pathnames. We support the usual operations to *create*, *delete*, *open*, *close*, *read*, and *write* files.

Moreover, GFS has *snapshot* and *record append* operations. Snapshot creates a copy of a file or a directory tree at low cost. Record append allows multiple clients to append data to the same file concurrently while guaranteeing the atomicity of each individual client's append. It is useful for implementing multi-way merge results and producer-consumer queues that many clients can simultaneously append to without additional locking. We have found these types of files to be invaluable in building large distributed applications. Snapshot and record append are discussed further in Sections 3.4 and 3.3 respectively.

2.3 Architecture

A GFS cluster consists of a single *master* and multiple *chunkservers* and is accessed by multiple *clients*, as shown in Figure 1. Each of these is typically a commodity Linux machine running a user-level server process. It is easy to run both a chunkserver and a client on the same machine, as long as machine resources permit and the lower reliability caused by running possibly flaky application code is acceptable.

Files are divided into fixed-size *chunks*. Each chunk is identified by an immutable and globally unique 64 bit *chunk handle* assigned by the master at the time of chunk creation. Chunkservers store chunks on local disks as Linux files and read or write chunk data specified by a chunk handle and byte range. For reliability, each chunk is replicated on multiple chunkservers. By default, we store three replicas, though users can designate different replication levels for different regions of the file namespace.

The master maintains all file system metadata. This includes the namespace, access control information, the mapping from files to chunks, and the current locations of chunks. It also controls system-wide activities such as chunk lease management, garbage collection of orphaned chunks, and chunk migration between chunkservers. The master periodically communicates with each chunkserver in *HeartBeat* messages to give it instructions and collect its state.

GFS client code linked into each application implements the file system API and communicates with the master and chunkservers to read or write data on behalf of the application. Clients interact with the master for metadata operations, but all data-bearing communication goes directly to the chunkservers. We do not provide the POSIX API and therefore need not hook into the Linux vnode layer.

Neither the client nor the chunkserver caches file data. Client caches offer little benefit because most applications stream through huge files or have working sets too large to be cached. Not having them simplifies the client and the overall system by eliminating cache coherence issues. (Clients do cache metadata, however.) Chunkservers need not cache file data because chunks are stored as local files and so Linux's buffer cache already keeps frequently accessed data in memory.

2.4 Single Master

Having a single master vastly simplifies our design and enables the master to make sophisticated chunk placement

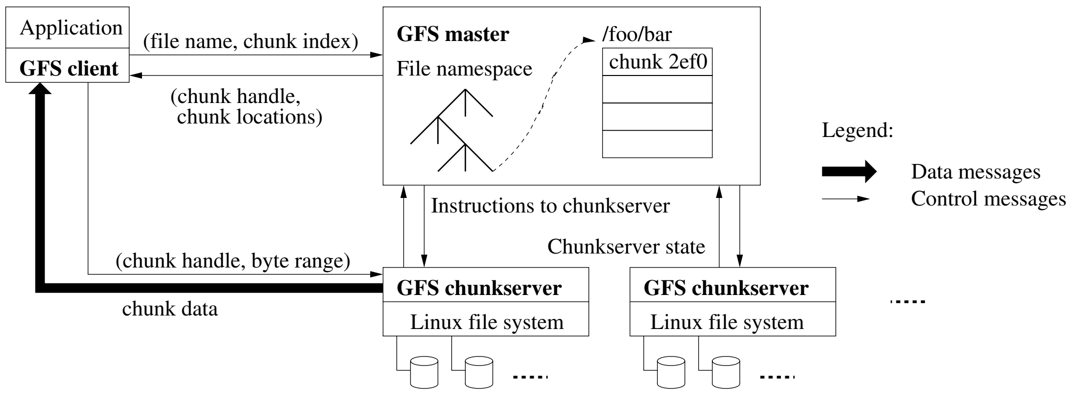


Figure 1: GFS Architecture

and replication decisions using global knowledge. However, we must minimize its involvement in reads and writes so that it does not become a bottleneck. Clients never read and write file data through the master. Instead, a client asks the master which chunkservers it should contact. It caches this information for a limited time and interacts with the chunkservers directly for many subsequent operations.

Let us explain the interactions for a simple read with reference to Figure 1. First, using the fixed chunk size, the client translates the file name and byte offset specified by the application into a chunk index within the file. Then, it sends the master a request containing the file name and chunk index. The master replies with the corresponding chunk handle and locations of the replicas. The client caches this information using the file name and chunk index as the key.

The client then sends a request to one of the replicas, most likely the closest one. The request specifies the chunk handle and a byte range within that chunk. Further reads of the same chunk require no more client-master interaction until the cached information expires or the file is reopened. In fact, the client typically asks for multiple chunks in the same request and the master can also include the information for chunks immediately following those requested. This extra information sidesteps several future client-master interactions at practically no extra cost.

2.5 Chunk Size

Chunk size is one of the key design parameters. We have chosen 64 MB, which is much larger than typical file system block sizes. Each chunk replica is stored as a plain Linux file on a chunkserver and is extended only as needed. Lazy space allocation avoids wasting space due to internal fragmentation, perhaps the greatest objection against such a large chunk size.

A large chunk size offers several important advantages. First, it reduces clients' need to interact with the master because reads and writes on the same chunk require only one initial request to the master for chunk location information. The reduction is especially significant for our workloads because applications mostly read and write large files sequentially. Even for small random reads, the client can comfortably cache all the chunk location information for a multi-TB working set. Second, since on a large chunk, a client is more likely to perform many operations on a given chunk, it can reduce network overhead by keeping a persistent TCP connection to the chunkserver over an extended period of time. Third, it reduces the size of the metadata stored on the master. This allows us to keep the metadata in memory, which in turn brings other advantages that we will discuss in Section 2.6.1.

On the other hand, a large chunk size, even with lazy space allocation, has its disadvantages. A small file consists of a small number of chunks, perhaps just one. The chunkservers storing those chunks may become hot spots if many clients are accessing the same file. In practice, hot spots have not been a major issue because our applications mostly read large multi-chunk files sequentially.

However, hot spots did develop when GFS was first used by a batch-queue system: an executable was written to GFS as a single-chunk file and then started on hundreds of machines at the same time. The few chunkservers storing this executable were overloaded by hundreds of simultaneous requests. We fixed this problem by storing such executables with a higher replication factor and by making the batch-queue system stagger application start times. A potential long-term solution is to allow clients to read data from other clients in such situations.

2.6 Metadata

The master stores three major types of metadata: the file and chunk namespaces, the mapping from files to chunks, and the locations of each chunk's replicas. All metadata is kept in the master's memory. The first two types (namespaces and file-to-chunk mapping) are also kept persistent by logging mutations to an *operation log* stored on the master's local disk and replicated on remote machines. Using a log allows us to update the master state simply, reliably, and without risking inconsistencies in the event of a master crash. The master does not store chunk location information persistently. Instead, it asks each chunkserver about its chunks at master startup and whenever a chunkserver joins the cluster.

2.6.1 In-Memory Data Structures

Since metadata is stored in memory, master operations are fast. Furthermore, it is easy and efficient for the master to periodically scan through its entire state in the background. This periodic scanning is used to implement chunk garbage collection, re-replication in the presence of chunkserver failures, and chunk migration to balance load and disk space usage across chunkservers. Sections 4.3 and 4.4 will discuss these activities further.

One potential concern for this memory-only approach is that the number of chunks and hence the capacity of the whole system is limited by how much memory the master has. This is not a serious limitation in practice. The master maintains less than 64 bytes of metadata for each 64 MB chunk. Most chunks are full because most files contain many chunks, only the last of which may be partially filled. Similarly, the file namespace data typically requires less then 64 bytes per file because it stores file names compactly using prefix compression.

If necessary to support even larger file systems, the cost of adding extra memory to the master is a small price to pay for the simplicity, reliability, performance, and flexibility we gain by storing the metadata in memory.

2.6.2 *Chunk Locations*

The master does not keep a persistent record of which chunkservers have a replica of a given chunk. It simply polls chunkservers for that information at startup. The master can keep itself up-to-date thereafter because it controls all chunk placement and monitors chunkserver status with regular *HeartBeat* messages.

We initially attempted to keep chunk location information persistently at the master, but we decided that it was much simpler to request the data from chunkservers at startup, and periodically thereafter. This eliminated the problem of keeping the master and chunkservers in sync as chunkservers join and leave the cluster, change names, fail, restart, and so on. In a cluster with hundreds of servers, these events happen all too often.

Another way to understand this design decision is to realize that a chunkserver has the final word over what chunks it does or does not have on its own disks. There is no point in trying to maintain a consistent view of this information on the master because errors on a chunkserver may cause chunks to vanish spontaneously (e.g., a disk may go bad and be disabled) or an operator may rename a chunkserver.

2.6.3 *Operation Log*

The operation log contains a historical record of critical metadata changes. It is central to GFS. Not only is it the only persistent record of metadata, but it also serves as a logical time line that defines the order of concurrent operations. Files and chunks, as well as their versions (see Section 4.5), are all uniquely and eternally identified by the logical times at which they were created.

Since the operation log is critical, we must store it reliably and not make changes visible to clients until metadata changes are made persistent. Otherwise, we effectively lose the whole file system or recent client operations even if the chunks themselves survive. Therefore, we replicate it on multiple remote machines and respond to a client operation only after flushing the corresponding log record to disk both locally and remotely. The master batches several log records together before flushing thereby reducing the impact of flushing and replication on overall system throughput.

The master recovers its file system state by replaying the operation log. To minimize startup time, we must keep the log small. The master checkpoints its state whenever the log grows beyond a certain size so that it can recover by loading the latest checkpoint from local disk and replaying only the limited number of log records after that. The checkpoint is in a compact B-tree like form that can be directly mapped into memory and used for namespace lookup without extra parsing. This further speeds up recovery and improves availability.

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

Table 1: File Region State After Mutation

Because building a checkpoint can take a while, the master's internal state is structured in such a way that a new checkpoint can be created without delaying incoming mutations. The master switches to a new log file and creates the new checkpoint in a separate thread. The new checkpoint includes all mutations before the switch. It can be created in a minute or so for a cluster with a few million files. When completed, it is written to disk both locally and remotely.

Recovery needs only the latest complete checkpoint and subsequent log files. Older checkpoints and log files can be freely deleted, though we keep a few around to guard against catastrophes. A failure during checkpointing does not affect correctness because the recovery code detects and skips incomplete checkpoints.

2.7 Consistency Model

GFS has a relaxed consistency model that supports our highly distributed applications well but remains relatively simple and efficient to implement. We now discuss GFS's guarantees and what they mean to applications. We also highlight how GFS maintains these guarantees but leave the details to other parts of the paper.

2.7.1 *Guarantees by GFS*

File namespace mutations (e.g., file creation) are atomic. They are handled exclusively by the master: namespace locking guarantees atomicity and correctness (Section 4.1); the master's operation log defines a global total order of these operations (Section 2.6.3).

The state of a file region after a data mutation depends on the type of mutation, whether it succeeds or fails, and whether there are concurrent mutations. Table 1 summarizes the result. A file region is *consistent* if all clients will always see the same data, regardless of which replicas they read from. A region is *defined* after a file data mutation if it is consistent and clients will see what the mutation writes in its entirety. When a mutation succeeds without interference from concurrent writers, the affected region is defined (and by implication consistent): all clients will always see what the mutation has written. Concurrent successful mutations leave the region undefined but consistent: all clients see the same data, but it may not reflect what any one mutation has written. Typically, it consists of mingled fragments from multiple mutations. A failed mutation makes the region inconsistent (hence also undefined): different clients may see different data at different times. We describe below how our applications can distinguish defined regions from undefined regions. The applications do not need to further distinguish between different kinds of undefined regions.

Data mutations may be *writes* or *record appends.* A write causes data to be written at an application-specified file offset. A record append causes data (the "record") to be appended *atomically at least once* even in the presence of concurrent mutations, but at an offset of GFS's choosing (Section 3.3). (In contrast, a "regular" append is merely a write at an offset that the client believes to be the current end of file.) The offset is returned to the client and marks the beginning of a defined region that contains the record. In addition, GFS may insert padding or record duplicates in between. They occupy regions considered to be inconsistent and are typically dwarfed by the amount of user data.

After a sequence of successful mutations, the mutated file region is guaranteed to be defined and contain the data written by the last mutation. GFS achieves this by (a) applying mutations to a chunk in the same order on all its replicas (Section 3.1), and (b) using chunk version numbers to detect any replica that has become stale because it has missed mutations while its chunkserver was down (Section 4.5). Stale replicas will never be involved in a mutation or given to clients asking the master for chunk locations. They are garbage collected at the earliest opportunity.

Since clients cache chunk locations, they may read from a stale replica before that information is refreshed. This window is limited by the cache entry's timeout and the next open of the file, which purges from the cache all chunk information for that file. Moreover, as most of our files are append-only, a stale replica usually returns a premature end of chunk rather than outdated data. When a reader retries and contacts the master, it will immediately get current chunk locations.

Long after a successful mutation, component failures can of course still corrupt or destroy data. GFS identifies failed chunkservers by regular handshakes between master and all chunkservers and detects data corruption by checksumming (Section 5.2). Once a problem surfaces, the data is restored from valid replicas as soon as possible (Section 4.3). A chunk is lost irreversibly only if all its replicas are lost before GFS can react, typically within minutes. Even in this case, it becomes unavailable, not corrupted: applications receive clear errors rather than corrupt data.

2.7.2 Implications for Applications

GFS applications can accommodate the relaxed consistency model with a few simple techniques already needed for other purposes: relying on appends rather than overwrites, checkpointing, and writing self-validating, self-identifying records.

Practically all our applications mutate files by appending rather than overwriting. In one typical use, a writer generates a file from beginning to end. It atomically renames the file to a permanent name after writing all the data, or periodically checkpoints how much has been successfully written. Checkpoints may also include application-level checksums. Readers verify and process only the file region up to the last checkpoint, which is known to be in the defined state. Regardless of consistency and concurrency issues, this approach has served us well. Appending is far more efficient and more resilient to application failures than random writes. Checkpointing allows writers to restart incrementally and keeps readers from processing successfully written file data that is still incomplete from the application's perspective.

In the other typical use, many writers concurrently append to a file for merged results or as a producer-consumer queue. Record append's append-at-least-once semantics preserves each writer's output. Readers deal with the occasional padding and duplicates as follows. Each record prepared by the writer contains extra information like checksums so that its validity can be verified. A reader can identify and discard extra padding and record fragments using the checksums. If it cannot tolerate the occasional duplicates (e.g., if they would trigger non-idempotent operations), it can filter them out using unique identifiers in the records, which are often needed anyway to name corresponding application entities such as web documents. These functionalities for record I/O (except duplicate removal) are in library code shared by our applications and applicable to other file interface implementations at Google. With that, the same sequence of records, plus rare duplicates, is always delivered to the record reader.

3. SYSTEM INTERACTIONS

We designed the system to minimize the master's involvement in all operations. With that background, we now describe how the client, master, and chunkservers interact to implement data mutations, atomic record append, and snapshot.

3.1 Leases and Mutation Order

A mutation is an operation that changes the contents or metadata of a chunk such as a write or an append operation. Each mutation is performed at all the chunk's replicas. We use leases to maintain a consistent mutation order across replicas. The master grants a chunk lease to one of the replicas, which we call the *primary.* The primary picks a serial order for all mutations to the chunk. All replicas follow this order when applying mutations. Thus, the global mutation order is defined first by the lease grant order chosen by the master, and within a lease by the serial numbers assigned by the primary.

The lease mechanism is designed to minimize management overhead at the master. A lease has an initial timeout of 60 seconds. However, as long as the chunk is being mutated, the primary can request and typically receive extensions from the master indefinitely. These extension requests and grants are piggybacked on the *HeartBeat* messages regularly exchanged between the master and all chunkservers. The master may sometimes try to revoke a lease before it expires (e.g., when the master wants to disable mutations on a file that is being renamed). Even if the master loses communication with a primary, it can safely grant a new lease to another replica after the old lease expires.

In Figure 2, we illustrate this process by following the control flow of a write through these numbered steps.

1. The client asks the master which chunkserver holds the current lease for the chunk and the locations of the other replicas. If no one has a lease, the master grants one to a replica it chooses (not shown).
2. The master replies with the identity of the primary and the locations of the other (*secondary*) replicas. The client caches this data for future mutations. It needs to contact the master again only when the primary

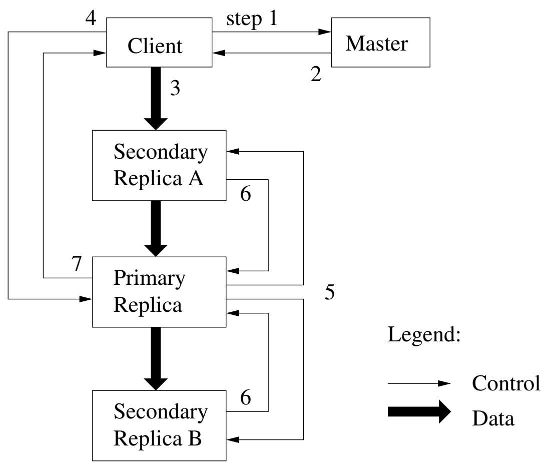


Figure 2: Write Control and Data Flow

becomes unreachable or replies that it no longer holds a lease.

3. The client pushes the data to all the replicas. A client can do so in any order. Each chunkserver will store the data in an internal LRU buffer cache until the data is used or aged out. By decoupling the data flow from the control flow, we can improve performance by scheduling the expensive data flow based on the network topology regardless of which chunkserver is the primary. Section 3.2 discusses this further.
4. Once all the replicas have acknowledged receiving the data, the client sends a write request to the primary. The request identifies the data pushed earlier to all of the replicas. The primary assigns consecutive serial numbers to all the mutations it receives, possibly from multiple clients, which provides the necessary serialization. It applies the mutation to its own local state in serial number order.
5. The primary forwards the write request to all secondary replicas. Each secondary replica applies mutations in the same serial number order assigned by the primary.
6. The secondaries all reply to the primary indicating that they have completed the operation.
7. The primary replies to the client. Any errors encountered at any of the replicas are reported to the client. In case of errors, the write may have succeeded at the primary and an arbitrary subset of the secondary replicas. (If it had failed at the primary, it would not have been assigned a serial number and forwarded.) The client request is considered to have failed, and the modified region is left in an inconsistent state. Our client code handles such errors by retrying the failed mutation. It will make a few attempts at steps (3) through (7) before falling back to a retry from the beginning of the write.

If a write by the application is large or straddles a chunk boundary, GFS client code breaks it down into multiple write operations. They all follow the control flow described above but may be interleaved with and overwritten by concurrent operations from other clients. Therefore, the shared file region may end up containing fragments from different clients, although the replicas will be identical because the individual operations are completed successfully in the same order on all replicas. This leaves the file region in consistent but undefined state as noted in Section 2.7.

3.2 Data Flow

We decouple the flow of data from the flow of control to use the network efficiently. While control flows from the client to the primary and then to all secondaries, data is pushed linearly along a carefully picked chain of chunkservers in a pipelined fashion. Our goals are to fully utilize each machine's network bandwidth, avoid network bottlenecks and high-latency links, and minimize the latency to push through all the data.

To fully utilize each machine's network bandwidth, the data is pushed linearly along a chain of chunkservers rather than distributed in some other topology (e.g., tree). Thus, each machine's full outbound bandwidth is used to transfer the data as fast as possible rather than divided among multiple recipients.

To avoid network bottlenecks and high-latency links (e.g., inter-switch links are often both) as much as possible, each machine forwards the data to the "closest" machine in the network topology that has not received it. Suppose the client is pushing data to chunkservers S1 through S4. It sends the data to the closest chunkserver, say S1. S1 forwards it to the closest chunkserver S2 through S4 closest to S1, say S2. Similarly, S2 forwards it to S3 or S4, whichever is closer to S2, and so on. Our network topology is simple enough that "distances" can be accurately estimated from IP addresses.

Finally, we minimize latency by pipelining the data transfer over TCP connections. Once a chunkserver receives some data, it starts forwarding immediately. Pipelining is especially helpful to us because we use a switched network with full-duplex links. Sending the data immediately does not reduce the receive rate. Without network congestion, the ideal elapsed time for transferring B bytes to R replicas is $B/T + RL$ where T is the network throughput and L is latency to transfer bytes between two machines. Our network links are typically 100 Mbps (T), and L is far below 1 ms. Therefore, 1 MB can ideally be distributed in about 80 ms.

3.3 Atomic Record Appends

GFS provides an atomic append operation called *record append*. In a traditional write, the client specifies the offset at which data is to be written. Concurrent writes to the same region are not serializable: the region may end up containing data fragments from multiple clients. In a record append, however, the client specifies only the data. GFS appends it to the file at least once atomically (i.e., as one continuous sequence of bytes) at an offset of GFS's choosing and returns that offset to the client. This is similar to writing to a file opened in `O_APPEND` mode in Unix without the race conditions when multiple writers do so concurrently.

Record append is heavily used by our distributed applications in which many clients on different machines append to the same file concurrently. Clients would need additional complicated and expensive synchronization, for example through a distributed lock manager, if they do so with traditional writes. In our workloads, such files often serve as multiple-producer/single-consumer queues or contain merged results from many different clients.

Record append is a kind of mutation and follows the control flow in Section 3.1 with only a little extra logic at the primary. The client pushes the data to all replicas of the last chunk of the file Then, it sends its request to the primary. The primary checks to see if appending the record to the current chunk would cause the chunk to exceed the maximum size (64 MB). If so, it pads the chunk to the maximum size, tells secondaries to do the same, and replies to the client indicating that the operation should be retried on the next chunk. (Record append is restricted to be at most one-fourth of the maximum chunk size to keep worst-case fragmentation at an acceptable level.) If the record fits within the maximum size, which is the common case, the primary appends the data to its replica, tells the secondaries to write the data at the exact offset where it has, and finally replies success to the client.

If a record append fails at any replica, the client retries the operation. As a result, replicas of the same chunk may contain different data possibly including duplicates of the same record in whole or in part. GFS does not guarantee that all replicas are bytewise identical. It only guarantees that the data is written at least once as an atomic unit. This property follows readily from the simple observation that for the operation to report success, the data must have been written at the same offset on all replicas of some chunk. Furthermore, after this, all replicas are at least as long as the end of record and therefore any future record will be assigned a higher offset or a different chunk even if a different replica later becomes the primary. In terms of our consistency guarantees, the regions in which successful record append operations have written their data are defined (hence consistent), whereas intervening regions are inconsistent (hence undefined). Our applications can deal with inconsistent regions as we discussed in Section 2.7.2.

3.4 Snapshot

The snapshot operation makes a copy of a file or a directory tree (the "source") almost instantaneously, while minimizing any interruptions of ongoing mutations. Our users use it to quickly create branch copies of huge data sets (and often copies of those copies, recursively), or to checkpoint the current state before experimenting with changes that can later be committed or rolled back easily.

Like AFS [5], we use standard copy-on-write techniques to implement snapshots. When the master receives a snapshot request, it first revokes any outstanding leases on the chunks in the files it is about to snapshot. This ensures that any subsequent writes to these chunks will require an interaction with the master to find the lease holder. This will give the master an opportunity to create a new copy of the chunk first.

After the leases have been revoked or have expired, the master logs the operation to disk. It then applies this log record to its in-memory state by duplicating the metadata for the source file or directory tree. The newly created snapshot files point to the same chunks as the source files.

The first time a client wants to write to a chunk C after the snapshot operation, it sends a request to the master to find the current lease holder. The master notices that the reference count for chunk C is greater than one. It defers replying to the client request and instead picks a new chunk handle C'. It then asks each chunkserver that has a current replica of C to create a new chunk called C'. By creating the new chunk on the same chunkservers as the original, we ensure that the data can be copied locally, not over the network (our disks are about three times as fast as our 100 Mb Ethernet links). From this point, request handling is no different from that for any chunk: the master grants one of the replicas a lease on the new chunk C' and replies to the client, which can write the chunk normally, not knowing that it has just been created from an existing chunk.

4. MASTER OPERATION

The master executes all namespace operations. In addition, it manages chunk replicas throughout the system: it makes placement decisions, creates new chunks and hence replicas, and coordinates various system-wide activities to keep chunks fully replicated, to balance load across all the chunkservers, and to reclaim unused storage. We now discuss each of these topics.

4.1 Namespace Management and Locking

Many master operations can take a long time: for example, a snapshot operation has to revoke chunkserver leases on all chunks covered by the snapshot. We do not want to delay other master operations while they are running. Therefore, we allow multiple operations to be active and use locks over regions of the namespace to ensure proper serialization.

Unlike many traditional file systems, GFS does not have a per-directory data structure that lists all the files in that directory. Nor does it support aliases for the same file or directory (i.e, hard or symbolic links in Unix terms). GFS logically represents its namespace as a lookup table mapping full pathnames to metadata. With prefix compression, this table can be efficiently represented in memory. Each node in the namespace tree (either an absolute file name or an absolute directory name) has an associated read-write lock.

Each master operation acquires a set of locks before it runs. Typically, if it involves `/d1/d2/.../dn/leaf`, it will acquire read-locks on the directory names `/d1`, `/d1/d2`, ..., `/d1/d2/.../dn`, and either a read lock or a write lock on the full pathname `/d1/d2/.../dn/leaf`. Note that `leaf` may be a file or directory depending on the operation.

We now illustrate how this locking mechanism can prevent a file `/home/user/foo` from being created while `/home/user` is being snapshotted to `/save/user`. The snapshot operation acquires read locks on `/home` and `/save`, and write locks on `/home/user` and `/save/user`. The file creation acquires read locks on `/home` and `/home/user`, and a write lock on `/home/user/foo`. The two operations will be serialized properly because they try to obtain conflicting locks on `/home/user`. File creation does not require a write lock on the parent directory because there is no "directory", or *inode*-like, data structure to be protected from modification. The read lock on the name is sufficient to protect the parent directory from deletion.

One nice property of this locking scheme is that it allows concurrent mutations in the same directory. For example, multiple file creations can be executed concurrently in the same directory: each acquires a read lock on the directory name and a write lock on the file name. The read lock on the directory name suffices to prevent the directory from being deleted, renamed, or snapshotted. The write locks on file names serialize attempts to create a file with the same name twice.

Since the namespace can have many nodes, read-write lock objects are allocated lazily and deleted once they are not in use. Also, locks are acquired in a consistent total order to prevent deadlock: they are first ordered by level in the namespace tree and lexicographically within the same level.

4.2 Replica Placement

A GFS cluster is highly distributed at more levels than one. It typically has hundreds of chunkservers spread across many machine racks. These chunkservers in turn may be accessed from hundreds of clients from the same or different racks. Communication between two machines on different racks may cross one or more network switches. Additionally, bandwidth into or out of a rack may be less than the aggregate bandwidth of all the machines within the rack. Multi-level distribution presents a unique challenge to distribute data for scalability, reliability, and availability.

The chunk replica placement policy serves two purposes: maximize data reliability and availability, and maximize network bandwidth utilization. For both, it is not enough to spread replicas across machines, which only guards against disk or machine failures and fully utilizes each machine's network bandwidth. We must also spread chunk replicas across racks. This ensures that some replicas of a chunk will survive and remain available even if an entire rack is damaged or offline (for example, due to failure of a shared resource like a network switch or power circuit). It also means that traffic, especially reads, for a chunk can exploit the aggregate bandwidth of multiple racks. On the other hand, write traffic has to flow through multiple racks, a tradeoff we make willingly.

4.3 Creation, Re-replication, Rebalancing

Chunk replicas are created for three reasons: chunk creation, re-replication, and rebalancing.

When the master *creates* a chunk, it chooses where to place the initially empty replicas. It considers several factors. (1) We want to place new replicas on chunkservers with below-average disk space utilization. Over time this will equalize disk utilization across chunkservers. (2) We want to limit the number of "recent" creations on each chunkserver. Although creation itself is cheap, it reliably predicts imminent heavy write traffic because chunks are created when demanded by writes, and in our append-once-read-many workload they typically become practically read-only once they have been completely written. (3) As discussed above, we want to spread replicas of a chunk across racks.

The master *re-replicates* a chunk as soon as the number of available replicas falls below a user-specified goal. This could happen for various reasons: a chunkserver becomes unavailable, it reports that its replica may be corrupted, one of its disks is disabled because of errors, or the replication goal is increased. Each chunk that needs to be re-replicated is prioritized based on several factors. One is how far it is from its replication goal. For example, we give higher priority to a chunk that has lost two replicas than to a chunk that has lost only one. In addition, we prefer to first re-replicate chunks for live files as opposed to chunks that belong to recently deleted files (see Section 4.4). Finally, to minimize the impact of failures on running applications, we boost the priority of any chunk that is blocking client progress.

The master picks the highest priority chunk and "clones" it by instructing some chunkserver to copy the chunk data directly from an existing valid replica. The new replica is placed with goals similar to those for creation: equalizing disk space utilization, limiting active clone operations on any single chunkserver, and spreading replicas across racks. To keep cloning traffic from overwhelming client traffic, the master limits the numbers of active clone operations both for the cluster and for each chunkserver. Additionally, each chunkserver limits the amount of bandwidth it spends on each clone operation by throttling its read requests to the source chunkserver.

Finally, the master *rebalances* replicas periodically: it examines the current replica distribution and moves replicas for better disk space and load balancing. Also through this process, the master gradually fills up a new chunkserver rather than instantly swamps it with new chunks and the heavy write traffic that comes with them. The placement criteria for the new replica are similar to those discussed above. In addition, the master must also choose which existing replica to remove. In general, it prefers to remove those on chunkservers with below-average free space so as to equalize disk space usage.

4.4 Garbage Collection

After a file is deleted, GFS does not immediately reclaim the available physical storage. It does so only lazily during regular garbage collection at both the file and chunk levels. We find that this approach makes the system much simpler and more reliable.

4.4.1 Mechanism

When a file is deleted by the application, the master logs the deletion immediately just like other changes. However instead of reclaiming resources immediately, the file is just renamed to a hidden name that includes the deletion timestamp. During the master's regular scan of the file system namespace, it removes any such hidden files if they have existed for more than three days (the interval is configurable). Until then, the file can still be read under the new, special name and can be undeleted by renaming it back to normal. When the hidden file is removed from the namespace, its in-memory metadata is erased. This effectively severs its links to all its chunks.

In a similar regular scan of the chunk namespace, the master identifies orphaned chunks (i.e., those not reachable from any file) and erases the metadata for those chunks. In a *HeartBeat* message regularly exchanged with the master, each chunkserver reports a subset of the chunks it has, and the master replies with the identity of all chunks that are no longer present in the master's metadata. The chunkserver is free to delete its replicas of such chunks.

4.4.2 Discussion

Although distributed garbage collection is a hard problem that demands complicated solutions in the context of programming languages, it is quite simple in our case. We can easily identify all references to chunks: they are in the file-to-chunk mappings maintained exclusively by the master. We can also easily identify all the chunk replicas: they are Linux files under designated directories on each chunkserver. Any such replica not known to the master is "garbage."

The garbage collection approach to storage reclamation offers several advantages over eager deletion. First, it is simple and reliable in a large-scale distributed system where component failures are common. Chunk creation may succeed on some chunkservers but not others, leaving replicas that the master does not know exist. Replica deletion messages may be lost, and the master has to remember to resend them across failures, both its own and the chunkserver's. Garbage collection provides a uniform and dependable way to clean up any replicas not known to be useful. Second, it merges storage reclamation into the regular background activities of the master, such as the regular scans of namespaces and handshakes with chunkservers. Thus, it is done in batches and the cost is amortized. Moreover, it is done only when the master is relatively free. The master can respond more promptly to client requests that demand timely attention. Third, the delay in reclaiming storage provides a safety net against accidental, irreversible deletion.

In our experience, the main disadvantage is that the delay sometimes hinders user effort to fine tune usage when storage is tight. Applications that repeatedly create and delete temporary files may not be able to reuse the storage right away. We address these issues by expediting storage reclamation if a deleted file is explicitly deleted again. We also allow users to apply different replication and reclamation policies to different parts of the namespace. For example, users can specify that all the chunks in the files within some directory tree are to be stored without replication, and any deleted files are immediately and irrevocably removed from the file system state.

4.5 Stale Replica Detection

Chunk replicas may become stale if a chunkserver fails and misses mutations to the chunk while it is down. For each chunk, the master maintains a *chunk version number* to distinguish between up-to-date and stale replicas.

Whenever the master grants a new lease on a chunk, it increases the chunk version number and informs the up-to-date replicas. The master and these replicas all record the new version number in their persistent state. This occurs before any client is notified and therefore before it can start writing to the chunk. If another replica is currently unavailable, its chunk version number will not be advanced. The master will detect that this chunkserver has a stale replica when the chunkserver restarts and reports its set of chunks and their associated version numbers. If the master sees a version number greater than the one in its records, the master assumes that it failed when granting the lease and so takes the higher version to be up-to-date.

The master removes stale replicas in its regular garbage collection. Before that, it effectively considers a stale replica not to exist at all when it replies to client requests for chunk information. As another safeguard, the master includes the chunk version number when it informs clients which chunkserver holds a lease on a chunk or when it instructs a chunkserver to read the chunk from another chunkserver in a cloning operation. The client or the chunkserver verifies the version number when it performs the operation so that it is always accessing up-to-date data.

5. FAULT TOLERANCE AND DIAGNOSIS

One of our greatest challenges in designing the system is dealing with frequent component failures. The quality and quantity of components together make these problems more the norm than the exception: we cannot completely trust the machines, nor can we completely trust the disks. Component failures can result in an unavailable system or, worse, corrupted data. We discuss how we meet these challenges and the tools we have built into the system to diagnose problems when they inevitably occur.

5.1 High Availability

Among hundreds of servers in a GFS cluster, some are bound to be unavailable at any given time. We keep the overall system highly available with two simple yet effective strategies: fast recovery and replication.

5.1.1 Fast Recovery

Both the master and the chunkserver are designed to restore their state and start in seconds no matter how they terminated. In fact, we do not distinguish between normal and abnormal termination; servers are routinely shut down just by killing the process. Clients and other servers experience a minor hiccup as they time out on their outstanding requests, reconnect to the restarted server, and retry. Section 6.2.2 reports observed startup times.

5.1.2 Chunk Replication

As discussed earlier, each chunk is replicated on multiple chunkservers on different racks. Users can specify different replication levels for different parts of the file namespace. The default is three. The master clones existing replicas as needed to keep each chunk fully replicated as chunkservers go offline or detect corrupted replicas through checksum verification (see Section 5.2). Although replication has served us well, we are exploring other forms of cross-server redundancy such as parity or erasure codes for our increasing read-only storage requirements. We expect that it is challenging but manageable to implement these more complicated redundancy schemes in our very loosely coupled system because our traffic is dominated by appends and reads rather than small random writes.

5.1.3 Master Replication

The master state is replicated for reliability. Its operation log and checkpoints are replicated on multiple machines. A mutation to the state is considered committed only after its log record has been flushed to disk locally and on all master replicas. For simplicity, one master process remains in charge of all mutations as well as background activities such as garbage collection that change the system internally. When it fails, it can restart almost instantly. If its machine or disk fails, monitoring infrastructure outside GFS starts a new master process elsewhere with the replicated operation log. Clients use only the canonical name of the master (e.g. gfs-test), which is a DNS alias that can be changed if the master is relocated to another machine.

Moreover, "shadow" masters provide read-only access to the file system even when the primary master is down. They are shadows, not mirrors, in that they may lag the primary slightly, typically fractions of a second. They enhance read availability for files that are not being actively mutated or applications that do not mind getting slightly stale results. In fact, since file content is read from chunkservers, applications do not observe stale file content. What could be stale within short windows is file metadata, like directory contents or access control information.

To keep itself informed, a shadow master reads a replica of the growing operation log and applies the same sequence of changes to its data structures exactly as the primary does. Like the primary, it polls chunkservers at startup (and infrequently thereafter) to locate chunk replicas and exchanges frequent handshake messages with them to monitor their status. It depends on the primary master only for replica location updates resulting from the primary's decisions to create and delete replicas.

5.2 Data Integrity

Each chunkserver uses checksumming to detect corruption of stored data. Given that a GFS cluster often has thousands of disks on hundreds of machines, it regularly experiences disk failures that cause data corruption or loss on both the read and write paths. (See Section 7 for one cause.) We can recover from corruption using other chunk replicas, but it would be impractical to detect corruption by comparing replicas across chunkservers. Moreover, divergent replicas may be legal: the semantics of GFS mutations, in particular atomic record append as discussed earlier, does not guarantee identical replicas. Therefore, each chunkserver must independently verify the integrity of its own copy by maintaining checksums.

A chunk is broken up into 64 KB blocks. Each has a corresponding 32 bit checksum. Like other metadata, checksums are kept in memory and stored persistently with logging, separate from user data.

For reads, the chunkserver verifies the checksum of data blocks that overlap the read range before returning any data to the requester, whether a client or another chunkserver. Therefore chunkservers will not propagate corruptions to other machines. If a block does not match the recorded checksum, the chunkserver returns an error to the requestor and reports the mismatch to the master. In response, the requestor will read from other replicas, while the master will clone the chunk from another replica. After a valid new replica is in place, the master instructs the chunkserver that reported the mismatch to delete its replica.

Checksumming has little effect on read performance for several reasons. Since most of our reads span at least a few blocks, we need to read and checksum only a relatively small amount of extra data for verification. GFS client code further reduces this overhead by trying to align reads at checksum block boundaries. Moreover, checksum lookups and comparison on the chunkserver are done without any I/O, and checksum calculation can often be overlapped with I/Os.

Checksum computation is heavily optimized for writes that append to the end of a chunk (as opposed to writes that overwrite existing data) because they are dominant in our workloads. We just incrementally update the checksum for the last partial checksum block, and compute new checksums for any brand new checksum blocks filled by the append. Even if the last partial checksum block is already corrupted and we fail to detect it now, the new checksum value will not match the stored data, and the corruption will be detected as usual when the block is next read.

In contrast, if a write overwrites an existing range of the chunk, we must read and verify the first and last blocks of the range being overwritten, then perform the write, and finally compute and record the new checksums. If we do not verify the first and last blocks before overwriting them partially, the new checksums may hide corruption that exists in the regions not being overwritten.

During idle periods, chunkservers can scan and verify the contents of inactive chunks. This allows us to detect corruption in chunks that are rarely read. Once the corruption is detected, the master can create a new uncorrupted replica and delete the corrupted replica. This prevents an inactive but corrupted chunk replica from fooling the master into thinking that it has enough valid replicas of a chunk.

5.3 Diagnostic Tools

Extensive and detailed diagnostic logging has helped immeasurably in problem isolation, debugging, and performance analysis, while incurring only a minimal cost. Without logs, it is hard to understand transient, non-repeatable interactions between machines. GFS servers generate diagnostic logs that record many significant events (such as chunkservers going up and down) and all RPC requests and replies. These diagnostic logs can be freely deleted without affecting the correctness of the system. However, we try to keep these logs around as far as space permits.

The RPC logs include the exact requests and responses sent on the wire, except for the file data being read or written. By matching requests with replies and collating RPC records on different machines, we can reconstruct the entire interaction history to diagnose a problem. The logs also serve as traces for load testing and performance analysis.

The performance impact of logging is minimal (and far outweighed by the benefits) because these logs are written sequentially and asynchronously. The most recent events are also kept in memory and available for continuous online monitoring.

6. MEASUREMENTS

In this section we present a few micro-benchmarks to illustrate the bottlenecks inherent in the GFS architecture and implementation, and also some numbers from real clusters in use at Google.

6.1 Micro-benchmarks

We measured performance on a GFS cluster consisting of one master, two master replicas, 16 chunkservers, and 16 clients. Note that this configuration was set up for ease of testing. Typical clusters have hundreds of chunkservers and hundreds of clients.

All the machines are configured with dual 1.4 GHz PIII processors, 2 GB of memory, two 80 GB 5400 rpm disks, and a 100 Mbps full-duplex Ethernet connection to an HP 2524 switch. All 19 GFS server machines are connected to one switch, and all 16 client machines to the other. The two switches are connected with a 1 Gbps link.

6.1.1 Reads

N clients read simultaneously from the file system. Each client reads a randomly selected 4 MB region from a 320 GB file set. This is repeated 256 times so that each client ends up reading 1 GB of data. The chunkservers taken together have only 32 GB of memory, so we expect at most a 10% hit rate in the Linux buffer cache. Our results should be close to cold cache results.

Figure 3(a) shows the aggregate read rate for N clients and its theoretical limit. The limit peaks at an aggregate of 125 MB/s when the 1 Gbps link between the two switches is saturated, or 12.5 MB/s per client when its 100 Mbps network interface gets saturated, whichever applies. The observed read rate is 10 MB/s, or 80% of the per-client limit, when just one client is reading. The aggregate read rate reaches 94 MB/s, about 75% of the 125 MB/s link limit, for 16 readers, or 6 MB/s per client. The efficiency drops from 80% to 75% because as the number of readers increases, so does the probability that multiple readers simultaneously read from the same chunkserver.

6.1.2 Writes

N clients write simultaneously to N distinct files. Each client writes 1 GB of data to a new file in a series of 1 MB writes. The aggregate write rate and its theoretical limit are shown in Figure 3(b). The limit plateaus at 67 MB/s because we need to write each byte to 3 of the 16 chunkservers, each with a 12.5 MB/s input connection.

The write rate for one client is 6.3 MB/s, about half of the limit. The main culprit for this is our network stack. It does not interact very well with the pipelining scheme we use for pushing data to chunk replicas. Delays in propagating data from one replica to another reduce the overall write rate.

Aggregate write rate reaches 35 MB/s for 16 clients (or 2.2 MB/s per client), about half the theoretical limit. As in the case of reads, it becomes more likely that multiple clients write concurrently to the same chunkserver as the number of clients increases. Moreover, collision is more likely for 16 writers than for 16 readers because each write involves three different replicas.

Writes are slower than we would like. In practice this has not been a major problem because even though it increases the latencies as seen by individual clients, it does not significantly affect the aggregate write bandwidth delivered by the system to a large number of clients.

6.1.3 Record Appends

Figure 3(c) shows record append performance. N clients append simultaneously to a single file. Performance is limited by the network bandwidth of the chunkservers that store the last chunk of the file, independent of the number of clients. It starts at 6.0 MB/s for one client and drops to 4.8 MB/s for 16 clients, mostly due to congestion and variances in network transfer rates seen by different clients.

Our applications tend to produce multiple such files concurrently. In other words, N clients append to M shared files simultaneously where both N and M are in the dozens or hundreds. Therefore, the chunkserver network congestion in our experiment is not a significant issue in practice because a client can make progress on writing one file while the chunkservers for another file are busy.

6.2 Real World Clusters

We now examine two clusters in use within Google that are representative of several others like them. Cluster A is used regularly for research and development by over a hundred engineers. A typical task is initiated by a human user and runs up to several hours. It reads through a few MBs to a few TBs of data, transforms or analyzes the data, and writes the results back to the cluster. Cluster B is primarily used for production data processing. The tasks last much longer and continuously generate and process multi-TB data sets with only occasional human intervention. In both cases, a single "task" consists of many processes on many machines reading and writing many files simultaneously.

Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

Table 2: Characteristics of two GFS clusters

6.2.1 Storage

As shown by the first five entries in the table, both clusters have hundreds of chunkservers, support many TBs of disk space, and are fairly but not completely full. "Used space" includes all chunk replicas. Virtually all files are replicated three times. Therefore, the clusters store 18 TB and 52 TB of file data respectively.

The two clusters have similar numbers of files, though B has a larger proportion of dead files, namely files which were deleted or replaced by a new version but whose storage have not yet been reclaimed. It also has more chunks because its files tend to be larger.

6.2.2 Metadata

The chunkservers in aggregate store tens of GBs of metadata, mostly the checksums for 64 KB blocks of user data. The only other metadata kept at the chunkservers is the chunk version number discussed in Section 4.5.

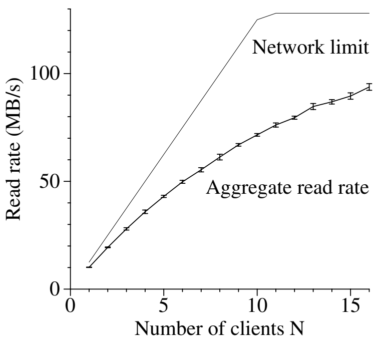
The metadata kept at the master is much smaller, only tens of MBs, or about 100 bytes per file on average. This agrees with our assumption that the size of the master's memory does not limit the system's capacity in practice. Most of the per-file metadata is the file names stored in a prefix-compressed form. Other metadata includes file ownership and permissions, mapping from files to chunks, and each chunk's current version. In addition, for each chunk we store the current replica locations and a reference count for implementing copy-on-write.

Each individual server, both chunkservers and the master, has only 50 to 100 MB of metadata. Therefore recovery is fast: it takes only a few seconds to read this metadata from disk before the server is able to answer queries. However, the master is somewhat hobbled for a period – typically 30 to 60 seconds – until it has fetched chunk location information from all chunkservers.

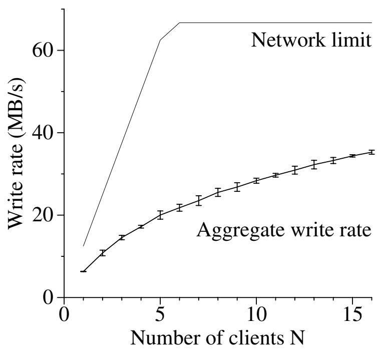
6.2.3 Read and Write Rates

Table 3 shows read and write rates for various time periods. Both clusters had been up for about one week when these measurements were taken. (The clusters had been restarted recently to upgrade to a new version of GFS.)

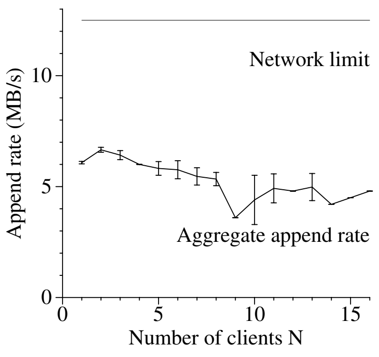
The average write rate was less than 30 MB/s since the restart. When we took these measurements, B was in the middle of a burst of write activity generating about 100 MB/s of data, which produced a 300 MB/s network load because writes are propagated to three replicas.



(a) Reads



(b) Writes



(c) Record appends

Figure 3: Aggregate Throughputs. Top curves show theoretical limits imposed by our network topology. Bottom curves show measured throughputs. They have error bars that show 95% confidence intervals, which are illegible in some cases because of low variance in measurements.

Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

Table 3: Performance Metrics for Two GFS Clusters

The read rates were much higher than the write rates. The total workload consists of more reads than writes as we have assumed. Both clusters were in the middle of heavy read activity. In particular, A had been sustaining a read rate of 580 MB/s for the preceding week. Its network configuration can support 750 MB/s, so it was using its resources efficiently. Cluster B can support peak read rates of 1300 MB/s, but its applications were using just 380 MB/s.

6.2.4 Master Load

Table 3 also shows that the rate of operations sent to the master was around 200 to 500 operations per second. The master can easily keep up with this rate, and therefore is not a bottleneck for these workloads.

In an earlier version of GFS, the master was occasionally a bottleneck for some workloads. It spent most of its time sequentially scanning through large directories (which contained hundreds of thousands of files) looking for particular files. We have since changed the master data structures to allow efficient binary searches through the namespace. It can now easily support many thousands of file accesses per second. If necessary, we could speed it up further by placing name lookup caches in front of the namespace data structures.

6.2.5 Recovery Time

After a chunkserver fails, some chunks will become underreplicated and must be cloned to restore their replication levels. The time it takes to restore all such chunks depends on the amount of resources. In one experiment, we killed a single chunkserver in cluster B. The chunkserver had about 15,000 chunks containing 600 GB of data. To limit the impact on running applications and provide leeway for scheduling decisions, our default parameters limit this cluster to 91 concurrent clonings (40% of the number of chunkservers) where each clone operation is allowed to consume at most 6.25 MB/s (50 Mbps). All chunks were restored in 23.2 minutes, at an effective replication rate of 440 MB/s.

In another experiment, we killed two chunkservers each with roughly 16,000 chunks and 660 GB of data. This double failure reduced 266 chunks to having a single replica. These 266 chunks were cloned at a higher priority, and were all restored to at least 2x replication within 2 minutes, thus putting the cluster in a state where it could tolerate another chunkserver failure without data loss.

6.3 Workload Breakdown

In this section, we present a detailed breakdown of the workloads on two GFS clusters comparable but not identical to those in Section 6.2. Cluster X is for research and development while cluster Y is for production data processing.

6.3.1 Methodology and Caveats

These results include only client originated requests so that they reflect the workload generated by our applications for the file system as a whole. They do not include interserver requests to carry out client requests or internal background activities, such as forwarded writes or rebalancing.

Statistics on I/O operations are based on information heuristically reconstructed from actual RPC requests logged by GFS servers. For example, GFS client code may break a read into multiple RPCs to increase parallelism, from which we infer the original read. Since our access patterns are highly stylized, we expect any error to be in the noise. Explicit logging by applications might have provided slightly more accurate data, but it is logistically impossible to recompile and restart thousands of running clients to do so and cumbersome to collect the results from as many machines.

One should be careful not to overly generalize from our workload. Since Google completely controls both GFS and its applications, the applications tend to be tuned for GFS, and conversely GFS is designed for these applications. Such mutual influence may also exist between general applications

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
0K	0.4	2.6	0	0	0	0
1B..1K	0.1	4.1	6.6	4.9	0.2	9.2
1K..8K	65.2	38.5	0.4	1.0	18.9	15.2
8K..64K	29.9	45.1	17.8	43.0	78.0	2.8
64K..128K	0.1	0.7	2.3	1.9	< .1	4.3
128K..256K	0.2	0.3	31.6	0.4	< .1	10.6
256K..512K	0.1	0.1	4.2	7.7	< .1	31.2
512K..1M	3.9	6.9	35.5	28.7	2.2	25.5
1M..inf	0.1	1.8	1.5	12.3	0.7	2.2

Table 4: Operations Breakdown by Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested.

and file systems, but the effect is likely more pronounced in our case.

6.3.2 Chunkserver Workload

Table 4 shows the distribution of operations by size. Read sizes exhibit a bimodal distribution. The small reads (under 64 KB) come from seek-intensive clients that look up small pieces of data within huge files. The large reads (over 512 KB) come from long sequential reads through entire files.

A significant number of reads return no data at all in cluster Y. Our applications, especially those in the production systems, often use files as producer-consumer queues. Producers append concurrently to a file while a consumer reads the end of file. Occasionally, no data is returned when the consumer outpaces the producers. Cluster X shows this less often because it is usually used for short-lived data analysis tasks rather than long-lived distributed applications.

Write sizes also exhibit a bimodal distribution. The large writes (over 256 KB) typically result from significant buffering within the writers. Writers that buffer less data, checkpoint or synchronize more often, or simply generate less data account for the smaller writes (under 64 KB).

As for record appends, cluster Y sees a much higher percentage of large record appends than cluster X does because our production systems, which use cluster Y, are more aggressively tuned for GFS.

Table 5 shows the total amount of data transferred in operations of various sizes. For all kinds of operations, the larger operations (over 256 KB) generally account for most of the bytes transferred. Small reads (under 64 KB) do transfer a small but significant portion of the read data because of the random seek workload.

6.3.3 Appends versus Writes

Record appends are heavily used especially in our production systems. For cluster X, the ratio of writes to record appends is 108:1 by bytes transferred and 8:1 by operation counts. For cluster Y, used by the production systems, the ratios are 3.7:1 and 2.5:1 respectively. Moreover, these ratios suggest that for both clusters record appends tend to be larger than writes. For cluster X, however, the overall usage of record append during the measured period is fairly low and so the results are likely skewed by one or two applications with particular buffer size choices.

As expected, our data mutation workload is dominated by appending rather than overwriting. We measured the amount of data overwritten on primary replicas. This approximates the case where a client deliberately overwrites previous written data rather than appends new data. For cluster X, overwriting accounts for under 0.0001% of bytes mutated and under 0.0003% of mutation operations. For cluster Y, the ratios are both 0.05%. Although this is minute, it is still higher than we expected. It turns out that most of these overwrites came from client retries due to errors or timeouts. They are not part of the workload *per se* but a consequence of the retry mechanism.

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
1B..1K	< .1	< .1	< .1	< .1	< .1	< .1
1K..8K	13.8	3.9	< .1	< .1	< .1	0.1
8K..64K	11.4	9.3	2.4	5.9	2.3	0.3
64K..128K	0.3	0.7	0.3	0.3	22.7	1.2
128K..256K	0.8	0.6	16.5	0.2	< .1	5.8
256K..512K	1.4	0.3	3.4	7.7	< .1	38.4
512K..1M	65.9	55.1	74.1	58.0	.1	46.8
1M..inf	6.4	30.1	3.3	28.0	53.9	7.4

Table 5: Bytes Transferred Breakdown by Operation Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested. The two may differ if the read attempts to read beyond end of file, which by design is not uncommon in our workloads.

Cluster	X	Y
Open	26.1	16.3
Delete	0.7	1.5
FindLocation	64.3	65.8
FindLeaseHolder	7.8	13.4
FindMatchingFiles	0.6	2.2
All other combined	0.5	0.8

Table 6: Master Requests Breakdown by Type (%)

6.3.4 Master Workload

Table 6 shows the breakdown by type of requests to the master. Most requests ask for chunk locations (*FindLocation*) for reads and lease holder information (*FindLeaseLocker*) for data mutations.

Clusters X and Y see significantly different numbers of *Delete* requests because cluster Y stores production data sets that are regularly regenerated and replaced with newer versions. Some of this difference is further hidden in the difference in *Open* requests because an old version of a file may be implicitly deleted by being opened for write from scratch (mode "w" in Unix open terminology).

FindMatchingFiles is a pattern matching request that supports "ls" and similar file system operations. Unlike other requests for the master, it may process a large part of the namespace and so may be expensive. Cluster Y sees it much more often because automated data processing tasks tend to examine parts of the file system to understand global application state. In contrast, cluster X's applications are under more explicit user control and usually know the names of all needed files in advance.

7. EXPERIENCES

In the process of building and deploying GFS, we have experienced a variety of issues, some operational and some technical.

Initially, GFS was conceived as the backend file system for our production systems. Over time, the usage evolved to include research and development tasks. It started with little support for things like permissions and quotas but now includes rudimentary forms of these. While production systems are well disciplined and controlled, users sometimes are not. More infrastructure is required to keep users from interfering with one another.

Some of our biggest problems were disk and Linux related. Many of our disks claimed to the Linux driver that they supported a range of IDE protocol versions but in fact responded reliably only to the more recent ones. Since the protocol versions are very similar, these drives mostly worked, but occasionally the mismatches would cause the drive and the kernel to disagree about the drive's state. This would corrupt data silently due to problems in the kernel. This problem motivated our use of checksums to detect data corruption, while concurrently we modified the kernel to handle these protocol mismatches.

Earlier we had some problems with Linux 2.2 kernels due to the cost of `fsync()`. Its cost is proportional to the size of the file rather than the size of the modified portion. This was a problem for our large operation logs especially before we implemented checkpointing. We worked around this for a time by using synchronous writes and eventually migrated to Linux 2.4.

Another Linux problem was a single reader-writer lock which any thread in an address space must hold when it pages in from disk (reader lock) or modifies the address space in an `mmap()` call (writer lock). We saw transient timeouts in our system under light load and looked hard for resource bottlenecks or sporadic hardware failures. Eventually, we found that this single lock blocked the primary network thread from mapping new data into memory while the disk threads were paging in previously mapped data. Since we are mainly limited by the network interface rather than by memory copy bandwidth, we worked around this by replacing `mmap()` with `pread()` at the cost of an extra copy.

Despite occasional problems, the availability of Linux code has helped us time and again to explore and understand system behavior. When appropriate, we improve the kernel and share the changes with the open source community.

8. RELATED WORK

Like other large distributed file systems such as AFS [5], GFS provides a location independent namespace which enables data to be moved transparently for load balance or fault tolerance. Unlike AFS, GFS spreads a file's data across storage servers in a way more akin to xFS [1] and Swift [3] in order to deliver aggregate performance and increased fault tolerance.

As disks are relatively cheap and replication is simpler than more sophisticated RAID [9] approaches, GFS currently uses only replication for redundancy and so consumes more raw storage than xFS or Swift.

In contrast to systems like AFS, xFS, Frangipani [12], and Intermezzo [6], GFS does not provide any caching below the file system interface. Our target workloads have little reuse within a single application run because they either stream through a large data set or randomly seek within it and read small amounts of data each time.

Some distributed file systems like Frangipani, xFS, Minnesota's GFS[11] and GPFS [10] remove the centralized server and rely on distributed algorithms for consistency and management. We opt for the centralized approach in order to simplify the design, increase its reliability, and gain flexibility. In particular, a centralized master makes it much easier to implement sophisticated chunk placement and replication policies since the master already has most of the relevant information and controls how it changes. We address fault tolerance by keeping the master state small and fully replicated on other machines. Scalability and high availability (for reads) are currently provided by our shadow master mechanism. Updates to the master state are made persistent by appending to a write-ahead log. Therefore we could adapt a primary-copy scheme like the one in Harp [7] to provide high availability with stronger consistency guarantees than our current scheme.

We are addressing a problem similar to Lustre [8] in terms of delivering aggregate performance to a large number of clients. However, we have simplified the problem significantly by focusing on the needs of our applications rather than building a POSIX-compliant file system. Additionally, GFS assumes large number of unreliable components and so fault tolerance is central to our design.

GFS most closely resembles the NASD architecture [4]. While the NASD architecture is based on network-attached disk drives, GFS uses commodity machines as chunkservers, as done in the NASD prototype. Unlike the NASD work, our chunkservers use lazily allocated fixed-size chunks rather than variable-length objects. Additionally, GFS implements features such as rebalancing, replication, and recovery that are required in a production environment.

Unlike Minnesota's GFS and NASD, we do not seek to alter the model of the storage device. We focus on addressing day-to-day data processing needs for complicated distributed systems with existing commodity components.

The producer-consumer queues enabled by atomic record appends address a similar problem as the distributed queues in River [2]. While River uses memory-based queues distributed across machines and careful data flow control, GFS uses a persistent file that can be appended to concurrently by many producers. The River model supports m-to-n distributed queues but lacks the fault tolerance that comes with persistent storage, while GFS only supports m-to-1 queues efficiently. Multiple consumers can read the same file, but they must coordinate to partition the incoming load.

9. CONCLUSIONS

The Google File System demonstrates the qualities essential for supporting large-scale data processing workloads on commodity hardware. While some design decisions are specific to our unique setting, many may apply to data processing tasks of a similar magnitude and cost consciousness.

We started by reexamining traditional file system assumptions in light of our current and anticipated application workloads and technological environment. Our observations have led to radically different points in the design space. We treat component failures as the norm rather than the exception, optimize for huge files that are mostly appended to (perhaps concurrently) and then read (usually sequentially), and both extend and relax the standard file system interface to improve the overall system.

Our system provides fault tolerance by constant monitoring, replicating crucial data, and fast and automatic recovery. Chunk replication allows us to tolerate chunkserver failures. The frequency of these failures motivated a novel online repair mechanism that regularly and transparently repairs the damage and compensates for lost replicas as soon as possible. Additionally, we use checksumming to detect data corruption at the disk or IDE subsystem level, which becomes all too common given the number of disks in the system.

Our design delivers high aggregate throughput to many concurrent readers and writers performing a variety of tasks. We achieve this by separating file system control, which passes through the master, from data transfer, which passes directly between chunkservers and clients. Master involvement in common operations is minimized by a large chunk size and by chunk leases, which delegates authority to primary replicas in data mutations. This makes possible a simple, centralized master that does not become a bottleneck. We believe that improvements in our networking stack will lift the current limitation on the write throughput seen by an individual client.

GFS has successfully met our storage needs and is widely used within Google as the storage platform for research and development as well as production data processing. It is an important tool that enables us to continue to innovate and attack problems on the scale of the entire web.

ACKNOWLEDGMENTS

We wish to thank the following people for their contributions to the system or the paper. Brain Bershad (our shepherd) and the anonymous reviewers gave us valuable comments and suggestions. Anurag Acharya, Jeff Dean, and David desJardins contributed to the early design. Fay Chang worked on comparison of replicas across chunkservers. Guy Edjlali worked on storage quota. Markus Gutschke worked on a testing framework and security enhancements. David Kramer worked on performance enhancements. Fay Chang, Urs Hoelzle, Max Ibel, Sharon Perl, Rob Pike, and Debby Wallach commented on earlier drafts of the paper. Many of our colleagues at Google bravely trusted their data to a new file system and gave us useful feedback. Yoshka helped with early testing.

REFERENCES

[1] Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randolph Wang. Serverless network file systems. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 109–126, Copper Mountain Resort, Colorado, December 1995.

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, Georgia, May 1999.

[3] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computer Systems*, 4(4):405–436, 1991.

[4] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th Architectural Support for Programming Languages and Operating Systems*, pages 92–103, San Jose, California, October 1998.

[5] John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, and Michael West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[6] InterMezzo. http://www.inter-mezzo.org, 2003.

[7] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *13th Symposium on Operating System Principles*, pages 226–238, Pacific Grove, CA, October 1991.

[8] Lustre. http://www.lustreorg, 2003.

[9] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, Illinois, September 1988.

[10] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies*, pages 231–244, Monterey, California, January 2002.

[11] Steven R. Soltis, Thomas M. Ruwart, and Matthew T. O'Keefe. The Gobal File System. In *Proceedings of the Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*, College Park, Maryland, September 1996.

[12] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 224–237, Saint-Malo, France, October 1997.

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 experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a 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* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis for a rewrite of our production indexing system. Section 7 discusses related and future work.

2 Programming Model

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the MapReduce library expresses the computation as two functions: *Map* and *Reduce*.

Map, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the *Reduce* function.

The *Reduce* function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per *Reduce* invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

2.1 Example

Consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code similar to the following pseudo-code:

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

The `map` function emits each word plus an associated count of occurrences (just '1' in this simple example). The `reduce` function sums together all counts emitted for a particular word.

In addition, the user writes code to fill in a *mapreduce specification* object with the names of the input and output files, and optional tuning parameters. The user then invokes the *MapReduce* function, passing it the specification object. The user's code is linked together with the MapReduce library (implemented in C++). Appendix A contains the full program text for this example.

2.2 Types

Even though the previous pseudo-code is written in terms of string inputs and outputs, conceptually the map and reduce functions supplied by the user have associated types:

map	`(k1,v1)`	$\rightarrow$ `list(k2,v2)`
reduce	`(k2,list(v2))`	$\rightarrow$ `list(v2)`

I.e., the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values.

Our C++ implementation passes strings to and from the user-defined functions and leaves it to the user code to convert between strings and appropriate types.

2.3 More Examples

Here are a few simple examples of interesting programs that can be easily expressed as MapReduce computations.

Distributed Grep: The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

Count of URL Access Frequency: The map function processes logs of web page requests and outputs $\langle$`URL,1`$\rangle$. The reduce function adds together all values for the same URL and emits a $\langle$`URL,total count`$\rangle$ pair.

Reverse Web-Link Graph: The map function outputs $\langle$`target,source`$\rangle$ pairs for each link to a `target` URL found in a page named `source`. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: $\langle$`target,`*list*`(source)`$\rangle$

Term-Vector per Host: A term vector summarizes the most important words that occur in a document or a set of documents as a list of $\langle word, frequency\rangle$ pairs. The map function emits a $\langle$`hostname,term vector`$\rangle$ pair for each input document (where the hostname is extracted from the URL of the document). The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final $\langle$`hostname,term vector`$\rangle$ pair.

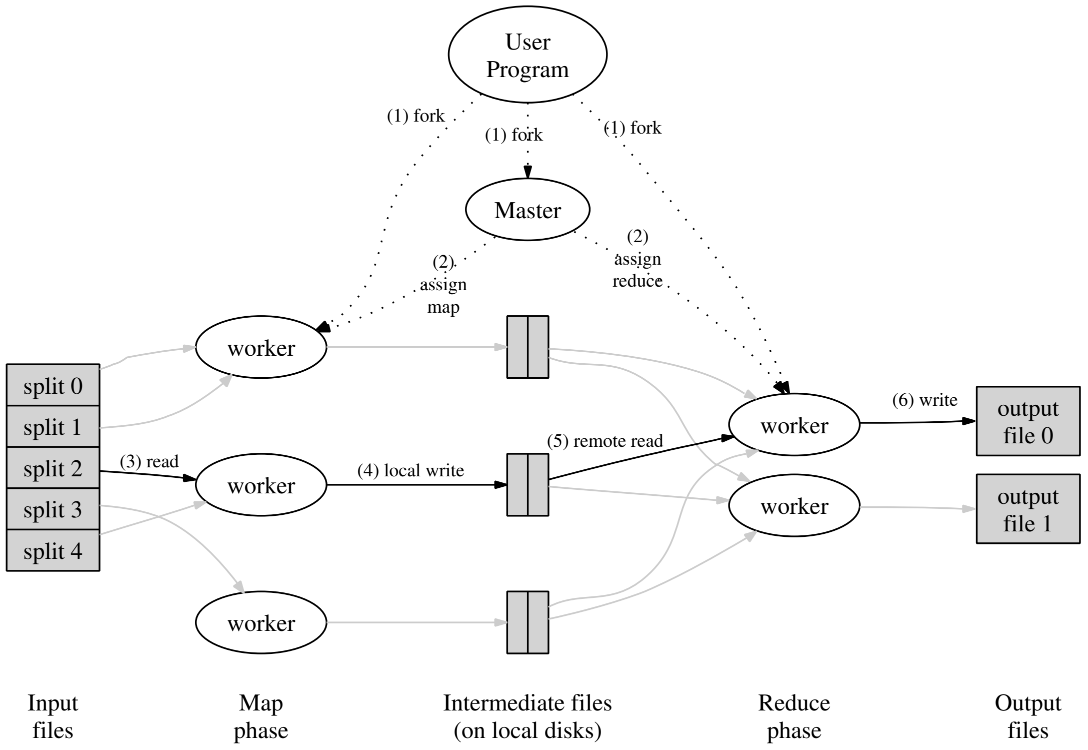


Figure 1: Execution overview

Inverted Index: The map function parses each document, and emits a sequence of $\langle$`word`, `document ID`$\rangle$ pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a $\langle$`word`, *list*(`document ID`)$\rangle$ pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

Distributed Sort: The map function extracts the key from each record, and emits a $\langle$`key`, `record`$\rangle$ pair. The reduce function emits all pairs unchanged. This computation depends on the partitioning facilities described in Section 4.1 and the ordering properties described in Section 4.2.

3 Implementation

Many different implementations of the MapReduce interface are possible. The right choice depends on the environment. For example, one implementation may be suitable for a small shared-memory machine, another for a large NUMA multi-processor, and yet another for an even larger collection of networked machines.

This section describes an implementation targeted to the computing environment in wide use at Google: large clusters of commodity PCs connected together with switched Ethernet [4]. In our environment:

(1) Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory per machine.

(2) Commodity networking hardware is used – typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth.

(3) A cluster consists of hundreds or thousands of machines, and therefore machine failures are common.

(4) Storage is provided by inexpensive IDE disks attached directly to individual machines. A distributed file system [8] developed in-house is used to manage the data stored on these disks. The file system uses replication to provide availability and reliability on top of unreliable hardware.

(5) Users submit jobs to a scheduling system. Each job consists of a set of tasks, and is mapped by the scheduler to a set of available machines within a cluster.

3.1 Execution Overview

The *Map* invocations are distributed across multiple machines by automatically partitioning the input data into a set of M *splits*. The input splits can be processed in parallel by different machines. *Reduce* invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., $hash(key)$ **mod** R). The number of partitions (R) and the partitioning function are specified by the user.

Figure 1 shows the overall flow of a MapReduce operation in our implementation. When the user program calls the `MapReduce` function, the following sequence of actions occurs (the numbered labels in Figure 1 correspond to the numbers in the list below):

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.

2. One of the copies of the program is special – the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.

3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined *Map* function. The intermediate key/value pairs produced by the *Map* function are buffered in memory.

4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's *Reduce* function. The output of the *Reduce* function is appended to a final output file for this reduce partition.

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the `MapReduce` call in the user program returns back to the user code.

After successful completion, the output of the mapreduce execution is available in the R output files (one per reduce task, with file names as specified by the user). Typically, users do not need to combine these R output files into one file – they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

3.2 Master Data Structures

The master keeps several data structures. For each map task and reduce task, it stores the state (*idle*, *in-progress*, or *completed*), and the identity of the worker machine (for non-idle tasks).

The master is the conduit through which the location of intermediate file regions is propagated from map tasks to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have *in-progress* reduce tasks.

3.3 Fault Tolerance

Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.

Worker Failure

The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial *idle* state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to *idle* and becomes eligible for rescheduling.

Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be re-executed since their output is stored in a global file system.

When a map task is executed first by worker A and then later executed by worker B (because A failed), all workers executing reduce tasks are notified of the re-execution. Any reduce task that has not already read the data from worker A will read the data from worker B.

MapReduce is resilient to large-scale worker failures. For example, during one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes. The MapReduce master simply re-executed the work done by the unreachable worker machines, and continued to make forward progress, eventually completing the MapReduce operation.

Master Failure

It is easy to make the master write periodic checkpoints of the master data structures described above. If the master task dies, a new copy can be started from the last checkpointed state. However, given that there is only a single master, its failure is unlikely; therefore our current implementation aborts the MapReduce computation if the master fails. Clients can check for this condition and retry the MapReduce operation if they desire.

Semantics in the Presence of Failures

When the user-supplied *map* and *reduce* operators are deterministic functions of their input values, our distributed implementation produces the same output as would have been produced by a non-faulting sequential execution of the entire program.

We rely on atomic commits of map and reduce task outputs to achieve this property. Each in-progress task writes its output to private temporary files. A reduce task produces one such file, and a map task produces R such files (one per reduce task). When a map task completes, the worker sends a message to the master and includes the names of the R temporary files in the message. If the master receives a completion message for an already completed map task, it ignores the message. Otherwise, it records the names of R files in a master data structure.

When a reduce task completes, the reduce worker atomically renames its temporary output file to the final output file. If the same reduce task is executed on multiple machines, multiple rename calls will be executed for the same final output file. We rely on the atomic rename operation provided by the underlying file system to guarantee that the final file system state contains just the data produced by one execution of the reduce task.

The vast majority of our *map* and *reduce* operators are deterministic, and the fact that our semantics are equivalent to a sequential execution in this case makes it very easy for programmers to reason about their program's behavior. When the *map* and/or *reduce* operators are non-deterministic, we provide weaker but still reasonable semantics. In the presence of non-deterministic operators, the output of a particular reduce task R_1 is equivalent to the output for R_1 produced by a sequential execution of the non-deterministic program. However, the output for a different reduce task R_2 may correspond to the output for R_2 produced by a different sequential execution of the non-deterministic program.

Consider map task M and reduce tasks R_1 and R_2. Let $e(R_i)$ be the execution of R_i that committed (there is exactly one such execution). The weaker semantics arise because $e(R_1)$ may have read the output produced by one execution of M and $e(R_2)$ may have read the output produced by a different execution of M.

3.4 Locality

Network bandwidth is a relatively scarce resource in our computing environment. We conserve network bandwidth by taking advantage of the fact that the input data (managed by GFS [8]) is stored on the local disks of the machines that make up our cluster. GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines. The MapReduce master takes the location information of the input files into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data. Failing that, it attempts to schedule a map task near a replica of that task's input data (e.g., on a worker machine that is on the same network switch as the machine containing the data). When running large MapReduce operations on a significant fraction of the workers in a cluster, most input data is read locally and consumes no network bandwidth.

3.5 Task Granularity

We subdivide the map phase into M pieces and the reduce phase into R pieces, as described above. Ideally, M and R should be much larger than the number of worker machines. Having each worker perform many different tasks improves dynamic load balancing, and also speeds up recovery when a worker fails: the many map tasks it has completed can be spread out across all the other worker machines.

There are practical bounds on how large M and R can be in our implementation, since the master must make $O(M + R)$ scheduling decisions and keeps $O(M * R)$ state in memory as described above. (The constant factors for memory usage are small however: the $O(M * R)$ piece of the state consists of approximately one byte of data per map task/reduce task pair.)

Furthermore, R is often constrained by users because the output of each reduce task ends up in a separate output file. In practice, we tend to choose M so that each individual task is roughly 16 MB to 64 MB of input data (so that the locality optimization described above is most effective), and we make R a small multiple of the number of worker machines we expect to use. We often perform MapReduce computations with $M = 200,000$ and $R = 5,000$, using 2,000 worker machines.

3.6 Backup Tasks

One of the common causes that lengthens the total time taken for a MapReduce operation is a "straggler": a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation. Stragglers can arise for a whole host of reasons. For example, a machine with a bad disk may experience frequent correctable errors that slow its read performance from 30 MB/s to 1 MB/s. The cluster scheduling system may have scheduled other tasks on the machine, causing it to execute the MapReduce code more slowly due to competition for CPU, memory, local disk, or network bandwidth. A recent problem we experienced was a bug in machine initialization code that caused processor caches to be disabled: computations on affected machines slowed down by over a factor of one hundred.

We have a general mechanism to alleviate the problem of stragglers. When a MapReduce operation is close to completion, the master schedules backup executions of the remaining *in-progress* tasks. The task is marked as completed whenever either the primary or the backup execution completes. We have tuned this mechanism so that it typically increases the computational resources used by the operation by no more than a few percent. We have found that this significantly reduces the time to complete large MapReduce operations. As an example, the sort program described in Section 5.3 takes 44% longer to complete when the backup task mechanism is disabled.

4 Refinements

Although the basic functionality provided by simply writing *Map* and *Reduce* functions is sufficient for most needs, we have found a few extensions useful. These are described in this section.

4.1 Partitioning Function

The users of MapReduce specify the number of reduce tasks/output files that they desire (R). Data gets partitioned across these tasks using a partitioning function on the intermediate key. A default partitioning function is provided that uses hashing (e.g. "$hash(key)$ **mod** R"). This tends to result in fairly well-balanced partitions. In some cases, however, it is useful to partition data by some other function of the key. For example, sometimes the output keys are URLs, and we want all entries for a single host to end up in the same output file. To support situations like this, the user of the MapReduce library can provide a special partitioning function. For example, using "$hash(Hostname(urlkey))$ **mod** R" as the partitioning function causes all URLs from the same host to end up in the same output file.

4.2 Ordering Guarantees

We guarantee that within a given partition, the intermediate key/value pairs are processed in increasing key order. This ordering guarantee makes it easy to generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key, or users of the output find it convenient to have the data sorted.

4.3 Combiner Function

In some cases, there is significant repetition in the intermediate keys produced by each map task, and the user-specified *Reduce* function is commutative and associative. A good example of this is the word counting example in Section 2.1. Since word frequencies tend to follow a Zipf distribution, each map task will produce hundreds or thousands of records of the form `<the, 1>`. All of these counts will be sent over the network to a single reduce task and then added together by the *Reduce* function to produce one number. We allow the user to specify an optional *Combiner* function that does partial merging of this data before it is sent over the network.

The *Combiner* function is executed on each machine that performs a map task. Typically the same code is used to implement both the combiner and the reduce functions. The only difference between a reduce function and a combiner function is how the MapReduce library handles the output of the function. The output of a reduce function is written to the final output file. The output of a combiner function is written to an intermediate file that will be sent to a reduce task.

Partial combining significantly speeds up certain classes of MapReduce operations. Appendix A contains an example that uses a combiner.

4.4 Input and Output Types

The MapReduce library provides support for reading input data in several different formats. For example, "text" mode input treats each line as a key/value pair: the key is the offset in the file and the value is the contents of the line. Another common supported format stores a sequence of key/value pairs sorted by key. Each input type implementation knows how to split itself into meaningful ranges for processing as separate map tasks (e.g. text mode's range splitting ensures that range splits occur only at line boundaries). Users can add support for a new input type by providing an implementation of a simple *reader* interface, though most users just use one of a small number of predefined input types.

A *reader* does not necessarily need to provide data read from a file. For example, it is easy to define a *reader* that reads records from a database, or from data structures mapped in memory.

In a similar fashion, we support a set of output types for producing data in different formats and it is easy for user code to add support for new output types.

4.5 Side-effects

In some cases, users of MapReduce have found it convenient to produce auxiliary files as additional outputs from their map and/or reduce operators. We rely on the application writer to make such side-effects atomic and idempotent. Typically the application writes to a temporary file and atomically renames this file once it has been fully generated.

We do not provide support for atomic two-phase commits of multiple output files produced by a single task. Therefore, tasks that produce multiple output files with cross-file consistency requirements should be deterministic. This restriction has never been an issue in practice.

4.6 Skipping Bad Records

Sometimes there are bugs in user code that cause the *Map* or *Reduce* functions to crash deterministically on certain records. Such bugs prevent a MapReduce operation from completing. The usual course of action is to fix the bug, but sometimes this is not feasible; perhaps the bug is in a third-party library for which source code is unavailable. Also, sometimes it is acceptable to ignore a few records, for example when doing statistical analysis on a large data set. We provide an optional mode of execution where the MapReduce library detects which records cause deterministic crashes and skips these records in order to make forward progress.

Each worker process installs a signal handler that catches segmentation violations and bus errors. Before invoking a user *Map* or *Reduce* operation, the MapReduce library stores the sequence number of the argument in a global variable. If the user code generates a signal, the signal handler sends a "last gasp" UDP packet that contains the sequence number to the MapReduce master. When the master has seen more than one failure on a particular record, it indicates that the record should be skipped when it issues the next re-execution of the corresponding Map or Reduce task.

4.7 Local Execution

Debugging problems in *Map* or *Reduce* functions can be tricky, since the actual computation happens in a distributed system, often on several thousand machines, with work assignment decisions made dynamically by the master. To help facilitate debugging, profiling, and small-scale testing, we have developed an alternative implementation of the MapReduce library that sequentially executes all of the work for a MapReduce operation on the local machine. Controls are provided to the user so that the computation can be limited to particular map tasks. Users invoke their program with a special flag and can then easily use any debugging or testing tools they find useful (e.g. `gdb`).

4.8 Status Information

The master runs an internal HTTP server and exports a set of status pages for human consumption. The status pages show the progress of the computation, such as how many tasks have been completed, how many are in progress, bytes of input, bytes of intermediate data, bytes of output, processing rates, etc. The pages also contain links to the standard error and standard output files generated by each task. The user can use this data to predict how long the computation will take, and whether or not more resources should be added to the computation. These pages can also be used to figure out when the computation is much slower than expected.

In addition, the top-level status page shows which workers have failed, and which map and reduce tasks they were processing when they failed. This information is useful when attempting to diagnose bugs in the user code.

4.9 Counters

The MapReduce library provides a counter facility to count occurrences of various events. For example, user code may want to count total number of words processed or the number of German documents indexed, etc.

To use this facility, user code creates a named counter object and then increments the counter appropriately in the *Map* and/or *Reduce* function. For example:

```
Counter* uppercase;
uppercase = GetCounter("uppercase");

map(String name, String contents):
  for each word w in contents:
    if (IsCapitalized(w)):
      uppercase->Increment();
    EmitIntermediate(w, "1");
```

The counter values from individual worker machines are periodically propagated to the master (piggybacked on the ping response). The master aggregates the counter values from successful map and reduce tasks and returns them to the user code when the MapReduce operation is completed. The current counter values are also displayed on the master status page so that a human can watch the progress of the live computation. When aggregating counter values, the master eliminates the effects of duplicate executions of the same map or reduce task to avoid double counting. (Duplicate executions can arise from our use of backup tasks and from re-execution of tasks due to failures.)

Some counter values are automatically maintained by the MapReduce library, such as the number of input key/value pairs processed and the number of output key/value pairs produced.

Users have found the counter facility useful for sanity checking the behavior of MapReduce operations. For example, in some MapReduce operations, the user code may want to ensure that the number of output pairs produced exactly equals the number of input pairs processed, or that the fraction of German documents processed is within some tolerable fraction of the total number of documents processed.

5 Performance

In this section we measure the performance of MapReduce on two computations running on a large cluster of machines. One computation searches through approximately one terabyte of data looking for a particular pattern. The other computation sorts approximately one terabyte of data.

These two programs are representative of a large subset of the real programs written by users of MapReduce – one class of programs shuffles data from one representation to another, and another class extracts a small amount of interesting data from a large data set.

5.1 Cluster Configuration

All of the programs were executed on a cluster that consisted of approximately 1800 machines. Each machine had two 2GHz Intel Xeon processors with Hyper-Threading enabled, 4GB of memory, two 160GB IDE disks, and a gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.

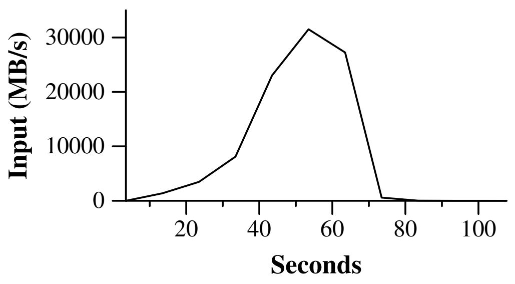


Figure 2: Data transfer rate over time

Out of the 4GB of memory, approximately 1-1.5GB was reserved by other tasks running on the cluster. The programs were executed on a weekend afternoon, when the CPUs, disks, and network were mostly idle.

5.2 Grep

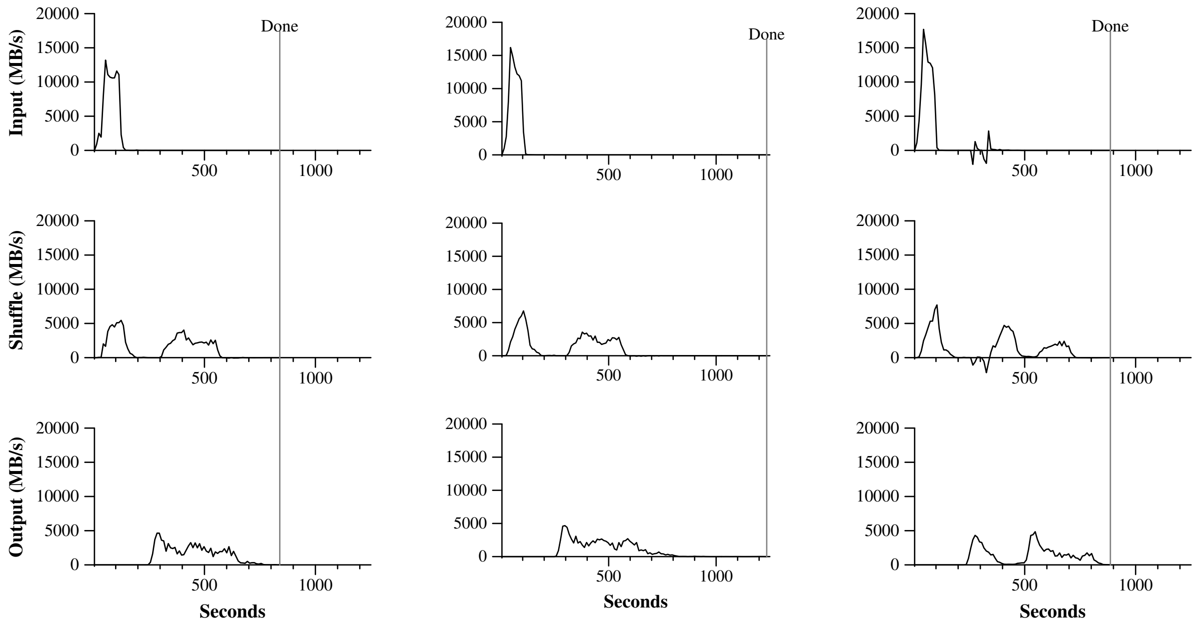
The *grep* program scans through 10^{10} 100-byte records, searching for a relatively rare three-character pattern (the pattern occurs in 92,337 records). The input is split into approximately 64MB pieces ($M = 15000$), and the entire output is placed in one file ($R = 1$).

Figure 2 shows the progress of the computation over time. The Y-axis shows the rate at which the input data is scanned. The rate gradually picks up as more machines are assigned to this MapReduce computation, and peaks at over 30 GB/s when 1764 workers have been assigned. As the map tasks finish, the rate starts dropping and hits zero about 80 seconds into the computation. The entire computation takes approximately 150 seconds from start to finish. This includes about a minute of startup overhead. The overhead is due to the propagation of the program to all worker machines, and delays interacting with GFS to open the set of 1000 input files and to get the information needed for the locality optimization.

5.3 Sort

The *sort* program sorts 10^{10} 100-byte records (approximately 1 terabyte of data). This program is modeled after the TeraSort benchmark [10].

The sorting program consists of less than 50 lines of user code. A three-line *Map* function extracts a 10-byte sorting key from a text line and emits the key and the



(a) Normal execution (b) No backup tasks (c) 200 tasks killed

Figure 3: Data transfer rates over time for different executions of the sort program

original text line as the intermediate key/value pair. We used a built-in *Identity* function as the *Reduce* operator. This functions passes the intermediate key/value pair unchanged as the output key/value pair. The final sorted output is written to a set of 2-way replicated GFS files (i.e., 2 terabytes are written as the output of the program).

As before, the input data is split into 64MB pieces ($M = 15000$). We partition the sorted output into 4000 files ($R = 4000$). The partitioning function uses the initial bytes of the key to segregate it into one of R pieces.

Our partitioning function for this benchmark has built-in knowledge of the distribution of keys. In a general sorting program, we would add a pre-pass MapReduce operation that would collect a sample of the keys and use the distribution of the sampled keys to compute split-points for the final sorting pass.

Figure 3 (a) shows the progress of a normal execution of the sort program. The top-left graph shows the rate at which input is read. The rate peaks at about 13 GB/s and dies off fairly quickly since all map tasks finish before 200 seconds have elapsed. Note that the input rate is less than for *grep*. This is because the sort map tasks spend about half their time and I/O bandwidth writing intermediate output to their local disks. The corresponding intermediate output for grep had negligible size.

The middle-left graph shows the rate at which data is sent over the network from the map tasks to the reduce tasks. This shuffling starts as soon as the first map task completes. The first hump in the graph is for the first batch of approximately 1700 reduce tasks (the entire MapReduce was assigned about 1700 machines, and each machine executes at most one reduce task at a time). Roughly 300 seconds into the computation, some of these first batch of reduce tasks finish and we start shuffling data for the remaining reduce tasks. All of the shuffling is done about 600 seconds into the computation.

The bottom-left graph shows the rate at which sorted data is written to the final output files by the reduce tasks. There is a delay between the end of the first shuffling period and the start of the writing period because the machines are busy sorting the intermediate data. The writes continue at a rate of about 2-4 GB/s for a while. All of the writes finish about 850 seconds into the computation. Including startup overhead, the entire computation takes 891 seconds. This is similar to the current best reported result of 1057 seconds for the TeraSort benchmark [18].

A few things to note: the input rate is higher than the shuffle rate and the output rate because of our locality optimization – most data is read from a local disk and bypasses our relatively bandwidth constrained network. The shuffle rate is higher than the output rate because the output phase writes two copies of the sorted data (we make two replicas of the output for reliability and availability reasons). We write two replicas because that is the mechanism for reliability and availability provided by our underlying file system. Network bandwidth requirements for writing data would be reduced if the underlying file system used erasure coding [14] rather than replication.

5.4 Effect of Backup Tasks

In Figure 3 (b), we show an execution of the sort program with backup tasks disabled. The execution flow is similar to that shown in Figure 3 (a), except that there is a very long tail where hardly any write activity occurs. After 960 seconds, all except 5 of the reduce tasks are completed. However these last few stragglers don't finish until 300 seconds later. The entire computation takes 1283 seconds, an increase of 44% in elapsed time.

5.5 Machine Failures

In Figure 3 (c), we show an execution of the sort program where we intentionally killed 200 out of 1746 worker processes several minutes into the computation. The underlying cluster scheduler immediately restarted new worker processes on these machines (since only the processes were killed, the machines were still functioning properly).

The worker deaths show up as a negative input rate since some previously completed map work disappears (since the corresponding map workers were killed) and needs to be redone. The re-execution of this map work happens relatively quickly. The entire computation finishes in 933 seconds including startup overhead (just an increase of 5% over the normal execution time).

6 Experience

We wrote the first version of the MapReduce library in February of 2003, and made significant enhancements to it in August of 2003, including the locality optimization, dynamic load balancing of task execution across worker machines, etc. Since that time, we have been pleasantly surprised at how broadly applicable the MapReduce library has been for the kinds of problems we work on. It has been used across a wide range of domains within Google, including:

- large-scale machine learning problems,
- clustering problems for the Google News and Froogle products,
- extraction of data used to produce reports of popular queries (e.g. Google Zeitgeist),
- extraction of properties of web pages for new experiments and products (e.g. extraction of geographical locations from a large corpus of web pages for localized search), and
- large-scale graph computations.

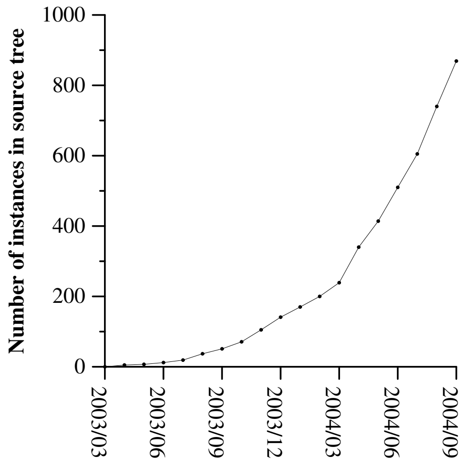


Figure 4: MapReduce instances over time

Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB
Average worker machines per job	157
Average worker deaths per job	1.2
Average map tasks per job	3,351
Average reduce tasks per job	55
Unique *map* implementations	395
Unique *reduce* implementations	269
Unique *map/reduce* combinations	426

Table 1: MapReduce jobs run in August 2004

Figure 4 shows the significant growth in the number of separate MapReduce programs checked into our primary source code management system over time, from 0 in early 2003 to almost 900 separate instances as of late September 2004. MapReduce has been so successful because it makes it possible to write a simple program and run it efficiently on a thousand machines in the course of half an hour, greatly speeding up the development and prototyping cycle. Furthermore, it allows programmers who have no experience with distributed and/or parallel systems to exploit large amounts of resources easily.

At the end of each job, the MapReduce library logs statistics about the computational resources used by the job. In Table 1, we show some statistics for a subset of MapReduce jobs run at Google in August 2004.

6.1 Large-Scale Indexing

One of our most significant uses of MapReduce to date has been a complete rewrite of the production indexing system that produces the data structures used for the Google web search service. The indexing system takes as input a large set of documents that have been retrieved by our crawling system, stored as a set of GFS files. The raw contents for these documents are more than 20 terabytes of data. The indexing process runs as a sequence of five to ten MapReduce operations. Using MapReduce (instead of the ad-hoc distributed passes in the prior version of the indexing system) has provided several benefits:

- The indexing code is simpler, smaller, and easier to understand, because the code that deals with fault tolerance, distribution and parallelization is hidden within the MapReduce library. For example, the size of one phase of the computation dropped from approximately 3800 lines of C++ code to approximately 700 lines when expressed using MapReduce.
- The performance of the MapReduce library is good enough that we can keep conceptually unrelated computations separate, instead of mixing them together to avoid extra passes over the data. This makes it easy to change the indexing process. For example, one change that took a few months to make in our old indexing system took only a few days to implement in the new system.
- The indexing process has become much easier to operate, because most of the problems caused by machine failures, slow machines, and networking hiccups are dealt with automatically by the MapReduce library without operator intervention. Furthermore, it is easy to improve the performance of the indexing process by adding new machines to the indexing cluster.

7 Related Work

Many systems have provided restricted programming models and used the restrictions to parallelize the computation automatically. For example, an associative function can be computed over all prefixes of an N element array in $\log N$ time on N processors using parallel prefix computations [6, 9, 13]. MapReduce can be considered a simplification and distillation of some of these models based on our experience with large real-world computations. More significantly, we provide a fault-tolerant implementation that scales to thousands of processors. In contrast, most of the parallel processing systems have only been implemented on smaller scales and leave the details of handling machine failures to the programmer.

Bulk Synchronous Programming [17] and some MPI primitives [11] provide higher-level abstractions that make it easier for programmers to write parallel programs. A key difference between these systems and MapReduce is that MapReduce exploits a restricted programming model to parallelize the user program automatically and to provide transparent fault-tolerance.

Our locality optimization draws its inspiration from techniques such as active disks [12, 15], where computation is pushed into processing elements that are close to local disks, to reduce the amount of data sent across I/O subsystems or the network. We run on commodity processors to which a small number of disks are directly connected instead of running directly on disk controller processors, but the general approach is similar.

Our backup task mechanism is similar to the eager scheduling mechanism employed in the Charlotte System [3]. One of the shortcomings of simple eager scheduling is that if a given task causes repeated failures, the entire computation fails to complete. We fix some instances of this problem with our mechanism for skipping bad records.

The MapReduce implementation relies on an in-house cluster management system that is responsible for distributing and running user tasks on a large collection of shared machines. Though not the focus of this paper, the cluster management system is similar in spirit to other systems such as Condor [16].

The sorting facility that is a part of the MapReduce library is similar in operation to NOW-Sort [1]. Source machines (map workers) partition the data to be sorted and send it to one of R reduce workers. Each reduce worker sorts its data locally (in memory if possible). Of course NOW-Sort does not have the user-definable Map and Reduce functions that make our library widely applicable.

River [2] provides a programming model where processes communicate with each other by sending data over distributed queues. Like MapReduce, the River system tries to provide good average case performance even in the presence of non-uniformities introduced by heterogeneous hardware or system perturbations. River achieves this by careful scheduling of disk and network transfers to achieve balanced completion times. MapReduce has a different approach. By restricting the programming model, the MapReduce framework is able to partition the problem into a large number of fine-grained tasks. These tasks are dynamically scheduled on available workers so that faster workers process more tasks. The restricted programming model also allows us to schedule redundant executions of tasks near the end of the job which greatly reduces completion time in the presence of non-uniformities (such as slow or stuck workers).

BAD-FS [5] has a very different programming model from MapReduce, and unlike MapReduce, is targeted to the execution of jobs across a wide-area network. However, there are two fundamental similarities. (1) Both systems use redundant execution to recover from data loss caused by failures. (2) Both use locality-aware scheduling to reduce the amount of data sent across congested network links.

TACC [7] is a system designed to simplify construction of highly-available networked services. Like MapReduce, it relies on re-execution as a mechanism for implementing fault-tolerance.

8 Conclusions

The MapReduce programming model has been successfully used at Google for many different purposes. We attribute this success to several reasons. First, the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing. Second, a large variety of problems are easily expressible as MapReduce computations. For example, MapReduce is used for the generation of data for Google's production web search service, for sorting, for data mining, for machine learning, and many other systems. Third, we have developed an implementation of MapReduce that scales to large clusters of machines comprising thousands of machines. The implementation makes efficient use of these machine resources and therefore is suitable for use on many of the large computational problems encountered at Google.

We have learned several things from this work. First, restricting the programming model makes it easy to parallelize and distribute computations and to make such computations fault-tolerant. Second, network bandwidth is a scarce resource. A number of optimizations in our system are therefore targeted at reducing the amount of data sent across the network: the locality optimization allows us to read data from local disks, and writing a single copy of the intermediate data to local disk saves network bandwidth. Third, redundant execution can be used to reduce the impact of slow machines, and to handle machine failures and data loss.

Acknowledgements

Josh Levenberg has been instrumental in revising and extending the user-level MapReduce API with a number of new features based on his experience with using MapReduce and other people's suggestions for enhancements. MapReduce reads its input from and writes its output to the Google File System [8]. We would like to thank Mohit Aron, Howard Gobioff, Markus Gutschke, David Kramer, Shun-Tak Leung, and Josh Redstone for their work in developing GFS. We would also like to thank Percy Liang and Olcan Sercinoglu for their work in developing the cluster management system used by MapReduce. Mike Burrows, Wilson Hsieh, Josh Levenberg, Sharon Perl, Rob Pike, and Debby Wallach provided helpful comments on earlier drafts of this paper. The anonymous OSDI reviewers, and our shepherd, Eric Brewer, provided many useful suggestions of areas where the paper could be improved. Finally, we thank all the users of MapReduce within Google's engineering organization for providing helpful feedback, suggestions, and bug reports.

References

[1] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, Georgia, May 1999.

[3] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.

[4] Luiz A. Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, April 2003.

[5] John Bent, Douglas Thain, Andrea C.Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation NSDI*, March 2004.

[6] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11), November 1989.

[7] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 78–91, Saint-Malo, France, 1997.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *19th Symposium on Operating Systems Principles*, pages 29–43, Lake George, New York, 2003.

[9] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.

[10] Jim Gray. Sort benchmark home page. http://research.microsoft.com/barc/SortBenchmark/.

[11] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[12] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the 2004 USENIX File and Storage Technologies FAST Conference*, April 2004.

[13] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[14] Michael O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[15] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *IEEE Computer*, pages 68–74, June 2001.

[16] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 2004.

[17] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1997.

[18] Jim Wyllie. Spsort: How to sort a terabyte quickly. http://alme1.almaden.ibm.com/cs/spsort.pdf.

A Word Frequency

This section contains a program that counts the number of occurrences of each unique word in a set of input files specified on the command line.

```
#include "mapreduce/mapreduce.h"

// User's map function
class WordCounter : public Mapper {
 public:
  virtual void Map(const MapInput& input) {
    const string& text = input.value();
    const int n = text.size();
    for (int i = 0; i < n; ) {
      // Skip past leading whitespace
      while ((i < n) && isspace(text[i]))
        i++;

      // Find word end
      int start = i;
      while ((i < n) && !isspace(text[i]))
        i++;
      if (start < i)
        Emit(text.substr(start,i-start),"1");
    }
  }
};
REGISTER_MAPPER(WordCounter);

// User's reduce function
class Adder : public Reducer {
  virtual void Reduce(ReduceInput* input) {
    // Iterate over all entries with the
    // same key and add the values
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }

    // Emit sum for input->key()
    Emit(IntToString(value));
  }
};
REGISTER_REDUCER(Adder);

int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);

  MapReduceSpecification spec;

  // Store list of input files into "spec"
  for (int i = 1; i < argc; i++) {
    MapReduceInput* input = spec.add_input();
    input->set_format("text");
    input->set_filepattern(argv[i]);
    input->set_mapper_class("WordCounter");
  }

  // Specify the output files:
  //    /gfs/test/freq-00000-of-00100
  //    /gfs/test/freq-00001-of-00100
  //    ...
  MapReduceOutput* out = spec.output();
  out->set_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_format("text");
  out->set_reducer_class("Adder");

  // Optional: do partial sums within map
  // tasks to save network bandwidth
  out->set_combiner_class("Adder");

  // Tuning parameters: use at most 2000
  // machines and 100 MB of memory per task
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_megabytes(100);

  // Now run it
  MapReduceResult result;
  if (!MapReduce(spec, &result)) abort();

  // Done: 'result' structure contains info
  // about counters, time taken, number of
  // machines used, etc.

  return 0;
}
```

Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

Google, Inc.

Abstract

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products. In this paper we describe the simple data model provided by Bigtable, which gives clients dynamic control over data layout and format, and we describe the design and implementation of Bigtable.

1 Introduction

Over the last two and a half years we have designed, implemented, and deployed a distributed storage system for managing structured data at Google called Bigtable. Bigtable is designed to reliably scale to petabytes of data and thousands of machines. Bigtable has achieved several goals: wide applicability, scalability, high performance, and high availability. Bigtable is used by more than sixty Google products and projects, including Google Analytics, Google Finance, Orkut, Personalized Search, Writely, and Google Earth. These products use Bigtable for a variety of demanding workloads, which range from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. The Bigtable clusters used by these products span a wide range of configurations, from a handful to thousands of servers, and store up to several hundred terabytes of data.

In many ways, Bigtable resembles a database: it shares many implementation strategies with databases. Parallel databases [14] and main-memory databases [13] have achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the data represented in the underlying storage. Data is indexed using row and column names that can be arbitrary strings. Bigtable also treats data as uninterpreted strings, although clients often serialize various forms of structured and semi-structured data into these strings. Clients can control the locality of their data through careful choices in their schemas. Finally, Bigtable schema parameters let clients dynamically control whether to serve data out of memory or from disk.

Section 2 describes the data model in more detail, and Section 3 provides an overview of the client API. Section 4 briefly describes the underlying Google infrastructure on which Bigtable depends. Section 5 describes the fundamentals of the Bigtable implementation, and Section 6 describes some of the refinements that we made to improve Bigtable's performance. Section 7 provides measurements of Bigtable's performance. We describe several examples of how Bigtable is used at Google in Section 8, and discuss some lessons we learned in designing and supporting Bigtable in Section 9. Finally, Section 10 describes related work, and Section 11 presents our conclusions.

2 Data Model

A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.

(row:string, column:string, time:int64) $\rightarrow$ string

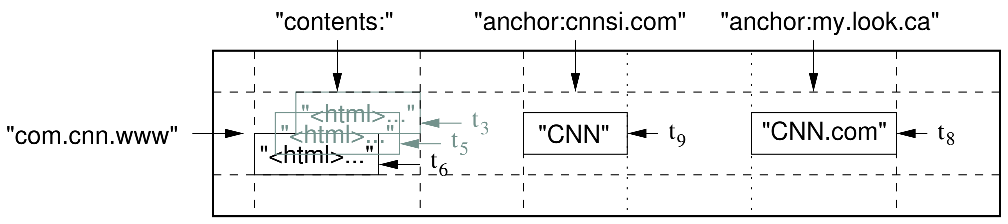


Figure 1: A slice of an example table that stores Web pages. The row name is a reversed URL. The contents column family contains the page contents, and the anchor column family contains the text of any anchors that reference the page. CNN's home page is referenced by both the Sports Illustrated and the MY-look home pages, so the row contains columns named anchor:cnnsi.com and anchor:my.look.ca. Each anchor cell has one version; the contents column has three versions, at timestamps t_3, t_5, and t_6.

We settled on this data model after examining a variety of potential uses of a Bigtable-like system. As one concrete example that drove some of our design decisions, suppose we want to keep a copy of a large collection of web pages and related information that could be used by many different projects; let us call this particular table the *Webtable*. In Webtable, we would use URLs as row keys, various aspects of web pages as column names, and store the contents of the web pages in the contents: column under the timestamps when they were fetched, as illustrated in Figure 1.

Rows

The row keys in a table are arbitrary strings (currently up to 64KB in size, although 10-100 bytes is a typical size for most of our users). Every read or write of data under a single row key is atomic (regardless of the number of different columns being read or written in the row), a design decision that makes it easier for clients to reason about the system's behavior in the presence of concurrent updates to the same row.

Bigtable maintains data in lexicographic order by row key. The row range for a table is dynamically partitioned. Each row range is called a *tablet*, which is the unit of distribution and load balancing. As a result, reads of short row ranges are efficient and typically require communication with only a small number of machines. Clients can exploit this property by selecting their row keys so that they get good locality for their data accesses. For example, in Webtable, pages in the same domain are grouped together into contiguous rows by reversing the hostname components of the URLs. For example, we store data for maps.google.com/index.html under the key com.google.maps/index.html. Storing pages from the same domain near each other makes some host and domain analyses more efficient.

Column Families

Column keys are grouped into sets called *column families*, which form the basic unit of access control. All data stored in a column family is usually of the same type (we compress data in the same column family together). A column family must be created before data can be stored under any column key in that family; after a family has been created, any column key within the family can be used. It is our intent that the number of distinct column families in a table be small (in the hundreds at most), and that families rarely change during operation. In contrast, a table may have an unbounded number of columns.

A column key is named using the following syntax: *family*:*qualifier*. Column family names must be printable, but qualifiers may be arbitrary strings. An example column family for the Webtable is language, which stores the language in which a web page was written. We use only one column key in the language family, and it stores each web page's language ID. Another useful column family for this table is anchor; each column key in this family represents a single anchor, as shown in Figure 1. The qualifier is the name of the referring site; the cell contents is the link text.

Access control and both disk and memory accounting are performed at the column-family level. In our Webtable example, these controls allow us to manage several different types of applications: some that add new base data, some that read the base data and create derived column families, and some that are only allowed to view existing data (and possibly not even to view all of the existing families for privacy reasons).

Timestamps

Each cell in a Bigtable can contain multiple versions of the same data; these versions are indexed by timestamp. Bigtable timestamps are 64-bit integers. They can be assigned by Bigtable, in which case they represent "real time" in microseconds, or be explicitly assigned by client

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

Figure 2: Writing to Bigtable.

applications. Applications that need to avoid collisions must generate unique timestamps themselves. Different versions of a cell are stored in decreasing timestamp order, so that the most recent versions can be read first.

To make the management of versioned data less onerous, we support two per-column-family settings that tell Bigtable to garbage-collect cell versions automatically. The client can specify either that only the last *n* versions of a cell be kept, or that only new-enough versions be kept (e.g., only keep values that were written in the last seven days).

In our Webtable example, we set the timestamps of the crawled pages stored in the contents: column to the times at which these page versions were actually crawled. The garbage-collection mechanism described above lets us keep only the most recent three versions of every page.

3 API

The Bigtable API provides functions for creating and deleting tables and column families. It also provides functions for changing cluster, table, and column family metadata, such as access control rights.

Client applications can write or delete values in Bigtable, look up values from individual rows, or iterate over a subset of the data in a table. Figure 2 shows C++ code that uses a `RowMutation` abstraction to perform a series of updates. (Irrelevant details were elided to keep the example short.) The call to `Apply` performs an atomic mutation to the Webtable: it adds one anchor to www.cnn.com and deletes a different anchor.

Figure 3 shows C++ code that uses a `Scanner` abstraction to iterate over all anchors in a particular row. Clients can iterate over multiple column families, and there are several mechanisms for limiting the rows, columns, and timestamps produced by a scan. For example, we could restrict the scan above to only produce anchors whose columns match the regular expression anchor:*.cnn.com, or to only produce anchors whose timestamps fall within ten days of the current time.

```
Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily("anchor");
stream->SetReturnAllVersions();
scanner.Lookup("com.cnn.www");
for (; !stream->Done(); stream->Next()) {
  printf("%s %s %lld %s\n",
         scanner.RowName(),
         stream->ColumnName(),
         stream->MicroTimestamp(),
         stream->Value());
}
```

Figure 3: Reading from Bigtable.

Bigtable supports several other features that allow the user to manipulate data in more complex ways. First, Bigtable supports single-row transactions, which can be used to perform atomic read-modify-write sequences on data stored under a single row key. Bigtable does not currently support general transactions across row keys, although it provides an interface for batching writes across row keys at the clients. Second, Bigtable allows cells to be used as integer counters. Finally, Bigtable supports the execution of client-supplied scripts in the address spaces of the servers. The scripts are written in a language developed at Google for processing data called Sawzall [28]. At the moment, our Sawzall-based API does not allow client scripts to write back into Bigtable, but it does allow various forms of data transformation, filtering based on arbitrary expressions, and summarization via a variety of operators.

Bigtable can be used with MapReduce [12], a framework for running large-scale parallel computations developed at Google. We have written a set of wrappers that allow a Bigtable to be used both as an input source and as an output target for MapReduce jobs.

4 Building Blocks

Bigtable is built on several other pieces of Google infrastructure. Bigtable uses the distributed Google File System (GFS) [17] to store log and data files. A Bigtable cluster typically operates in a shared pool of machines that run a wide variety of other distributed applications, and Bigtable processes often share the same machines with processes from other applications. Bigtable depends on a cluster management system for scheduling jobs, managing resources on shared machines, dealing with machine failures, and monitoring machine status.

The Google *SSTable* file format is used internally to store Bigtable data. An SSTable provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings. Operations are provided to look up the value associated with a specified key, and to iterate over all key/value pairs in a specified key range. Internally, each SSTable contains a sequence of blocks (typically each block is 64KB in size, but this is configurable). A block index (stored at the end of the SSTable) is used to locate blocks; the index is loaded into memory when the SSTable is opened. A lookup can be performed with a single disk seek: we first find the appropriate block by performing a binary search in the in-memory index, and then reading the appropriate block from disk. Optionally, an SSTable can be completely mapped into memory, which allows us to perform lookups and scans without touching disk.

Bigtable relies on a highly-available and persistent distributed lock service called Chubby [8]. A Chubby service consists of five active replicas, one of which is elected to be the master and actively serve requests. The service is live when a majority of the replicas are running and can communicate with each other. Chubby uses the Paxos algorithm [9, 23] to keep its replicas consistent in the face of failure. Chubby provides a namespace that consists of directories and small files. Each directory or file can be used as a lock, and reads and writes to a file are atomic. The Chubby client library provides consistent caching of Chubby files. Each Chubby client maintains a *session* with a Chubby service. A client's session expires if it is unable to renew its session lease within the lease expiration time. When a client's session expires, it loses any locks and open handles. Chubby clients can also register callbacks on Chubby files and directories for notification of changes or session expiration.

Bigtable uses Chubby for a variety of tasks: to ensure that there is at most one active master at any time; to store the bootstrap location of Bigtable data (see Section 5.1); to discover tablet servers and finalize tablet server deaths (see Section 5.2); to store Bigtable schema information (the column family information for each table); and to store access control lists. If Chubby becomes unavailable for an extended period of time, Bigtable becomes unavailable. We recently measured this effect in 14 Bigtable clusters spanning 11 Chubby instances. The average percentage of Bigtable server hours during which some data stored in Bigtable was not available due to Chubby unavailability (caused by either Chubby outages or network issues) was 0.0047%. The percentage for the single cluster that was most affected by Chubby unavailability was 0.0326%.

5 Implementation

The Bigtable implementation has three major components: a library that is linked into every client, one master server, and many tablet servers. Tablet servers can be dynamically added (or removed) from a cluster to accomodate changes in workloads.

The master is responsible for assigning tablets to tablet servers, detecting the addition and expiration of tablet servers, balancing tablet-server load, and garbage collection of files in GFS. In addition, it handles schema changes such as table and column family creations.

Each tablet server manages a set of tablets (typically we have somewhere between ten to a thousand tablets per tablet server). The tablet server handles read and write requests to the tablets that it has loaded, and also splits tablets that have grown too large.

As with many single-master distributed storage systems [17, 21], client data does not move through the master: clients communicate directly with tablet servers for reads and writes. Because Bigtable clients do not rely on the master for tablet location information, most clients never communicate with the master. As a result, the master is lightly loaded in practice.

A Bigtable cluster stores a number of tables. Each table consists of a set of tablets, and each tablet contains all data associated with a row range. Initially, each table consists of just one tablet. As a table grows, it is automatically split into multiple tablets, each approximately 100-200 MB in size by default.

5.1 Tablet Location

We use a three-level hierarchy analogous to that of a B$^+$-tree [10] to store tablet location information (Figure 4).

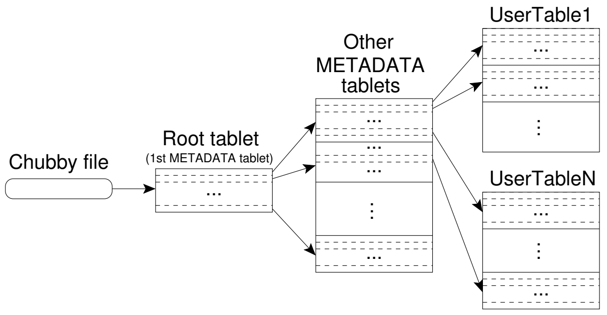


Figure 4: Tablet location hierarchy.

The first level is a file stored in Chubby that contains the location of the *root tablet*. The *root tablet* contains the location of all tablets in a special `METADATA` table. Each `METADATA` tablet contains the location of a set of user tablets. The *root tablet* is just the first tablet in the `METADATA` table, but is treated specially—it is never split—to ensure that the tablet location hierarchy has no more than three levels.

The `METADATA` table stores the location of a tablet under a row key that is an encoding of the tablet's table identifier and its end row. Each METADATA row stores approximately 1KB of data in memory. With a modest limit of 128 MB METADATA tablets, our three-level location scheme is sufficient to address 2^{34} tablets (or 2^{61} bytes in 128 MB tablets).

The client library caches tablet locations. If the client does not know the location of a tablet, or if it discovers that cached location information is incorrect, then it recursively moves up the tablet location hierarchy. If the client's cache is empty, the location algorithm requires three network round-trips, including one read from Chubby. If the client's cache is stale, the location algorithm could take up to six round-trips, because stale cache entries are only discovered upon misses (assuming that METADATA tablets do not move very frequently). Although tablet locations are stored in memory, so no GFS accesses are required, we further reduce this cost in the common case by having the client library prefetch tablet locations: it reads the metadata for more than one tablet whenever it reads the METADATA table.

We also store secondary information in the METADATA table, including a log of all events pertaining to each tablet (such as when a server begins serving it). This information is helpful for debugging and performance analysis.

5.2 Tablet Assignment

Each tablet is assigned to one tablet server at a time. The master keeps track of the set of live tablet servers, and the current assignment of tablets to tablet servers, including which tablets are unassigned. When a tablet is unassigned, and a tablet server with sufficient room for the tablet is available, the master assigns the tablet by sending a tablet load request to the tablet server.

Bigtable uses Chubby to keep track of tablet servers. When a tablet server starts, it creates, and acquires an exclusive lock on, a uniquely-named file in a specific Chubby directory. The master monitors this directory (the *servers directory*) to discover tablet servers. A tablet server stops serving its tablets if it loses its exclusive lock: e.g., due to a network partition that caused the server to lose its Chubby session. (Chubby provides an efficient mechanism that allows a tablet server to check whether it still holds its lock without incurring network traffic.) A tablet server will attempt to reacquire an exclusive lock on its file as long as the file still exists. If the file no longer exists, then the tablet server will never be able to serve again, so it kills itself. Whenever a tablet server terminates (e.g., because the cluster management system is removing the tablet server's machine from the cluster), it attempts to release its lock so that the master will reassign its tablets more quickly.

The master is responsible for detecting when a tablet server is no longer serving its tablets, and for reassigning those tablets as soon as possible. To detect when a tablet server is no longer serving its tablets, the master periodically asks each tablet server for the status of its lock. If a tablet server reports that it has lost its lock, or if the master was unable to reach a server during its last several attempts, the master attempts to acquire an exclusive lock on the server's file. If the master is able to acquire the lock, then Chubby is live and the tablet server is either dead or having trouble reaching Chubby, so the master ensures that the tablet server can never serve again by deleting its server file. Once a server's file has been deleted, the master can move all the tablets that were previously assigned to that server into the set of unassigned tablets. To ensure that a Bigtable cluster is not vulnerable to networking issues between the master and Chubby, the master kills itself if its Chubby session expires. However, as described above, master failures do not change the assignment of tablets to tablet servers.

When a master is started by the cluster management system, it needs to discover the current tablet assignments before it can change them. The master executes the following steps at startup. (1) The master grabs a unique *master* lock in Chubby, which prevents concurrent master instantiations. (2) The master scans the servers directory in Chubby to find the live servers. (3) The master communicates with every live tablet server to discover what tablets are already assigned to each server. (4) The master scans the METADATA table to learn the set of tablets. Whenever this scan encounters a tablet that is not already assigned, the master adds the tablet to the set of unassigned tablets, which makes the tablet eligible for tablet assignment.

One complication is that the scan of the METADATA table cannot happen until the METADATA tablets have been assigned. Therefore, before starting this scan (step 4), the master adds the root tablet to the set of unassigned tablets if an assignment for the root tablet was not discovered during step 3. This addition ensures that the root tablet will be assigned. Because the root tablet contains the names of all METADATA tablets, the master knows about all of them after it has scanned the root tablet.

The set of existing tablets only changes when a table is created or deleted, two existing tablets are merged to form one larger tablet, or an existing tablet is split into two smaller tablets. The master is able to keep track of these changes because it initiates all but the last. Tablet splits are treated specially since they are initiated by a tablet server. The tablet server commits the split by recording information for the new tablet in the METADATA table. When the split has committed, it notifies the master. In case the split notification is lost (either because the tablet server or the master died), the master detects the new tablet when it asks a tablet server to load the tablet that has now split. The tablet server will notify the master of the split, because the tablet entry it finds in the METADATA table will specify only a portion of the tablet that the master asked it to load.

5.3 Tablet Serving

The persistent state of a tablet is stored in GFS, as illustrated in Figure 5. Updates are committed to a commit log that stores redo records. Of these updates, the recently committed ones are stored in memory in a sorted buffer called a *memtable*; the older updates are stored in a sequence of SSTables. To recover a tablet, a tablet server

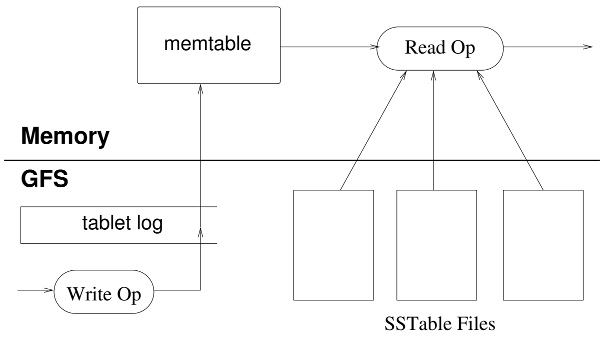


Figure 5: Tablet Representation

reads its metadata from the METADATA table. This metadata contains the list of SSTables that comprise a tablet and a set of a redo points, which are pointers into any commit logs that may contain data for the tablet. The server reads the indices of the SSTables into memory and reconstructs the memtable by applying all of the updates that have committed since the redo points.

When a write operation arrives at a tablet server, the server checks that it is well-formed, and that the sender is authorized to perform the mutation. Authorization is performed by reading the list of permitted writers from a Chubby file (which is almost always a hit in the Chubby client cache). A valid mutation is written to the commit log. Group commit is used to improve the throughput of lots of small mutations [13, 16]. After the write has been committed, its contents are inserted into the memtable.

When a read operation arrives at a tablet server, it is similarly checked for well-formedness and proper authorization. A valid read operation is executed on a merged view of the sequence of SSTables and the memtable. Since the SSTables and the memtable are lexicographically sorted data structures, the merged view can be formed efficiently.

Incoming read and write operations can continue while tablets are split and merged.

5.4 Compactions

As write operations execute, the size of the memtable increases. When the memtable size reaches a threshold, the memtable is frozen, a new memtable is created, and the frozen memtable is converted to an SSTable and written to GFS. This *minor compaction* process has two goals: it shrinks the memory usage of the tablet server, and it reduces the amount of data that has to be read from the commit log during recovery if this server dies. Incoming read and write operations can continue while compactions occur.

Every minor compaction creates a new SSTable. If this behavior continued unchecked, read operations might need to merge updates from an arbitrary number of SSTables. Instead, we bound the number of such files by periodically executing a *merging compaction* in the background. A merging compaction reads the contents of a few SSTables and the memtable, and writes out a new SSTable. The input SSTables and memtable can be discarded as soon as the compaction has finished.

A merging compaction that rewrites all SSTables into exactly one SSTable is called a *major compaction*. SSTables produced by non-major compactions can contain special deletion entries that suppress deleted data in older SSTables that are still live. A major compaction, on the other hand, produces an SSTable that contains no deletion information or deleted data. Bigtable cycles through all of its tablets and regularly applies major compactions to them. These major compactions allow Bigtable to reclaim resources used by deleted data, and also allow it to ensure that deleted data disappears from the system in a timely fashion, which is important for services that store sensitive data.

6 Refinements

The implementation described in the previous section required a number of refinements to achieve the high performance, availability, and reliability required by our users. This section describes portions of the implementation in more detail in order to highlight these refinements.

Locality groups

Clients can group multiple column families together into a *locality group*. A separate SSTable is generated for each locality group in each tablet. Segregating column families that are not typically accessed together into separate locality groups enables more efficient reads. For example, page metadata in Webtable (such as language and checksums) can be in one locality group, and the contents of the page can be in a different group: an application that wants to read the metadata does not need to read through all of the page contents.

In addition, some useful tuning parameters can be specified on a per-locality group basis. For example, a locality group can be declared to be in-memory. SSTables for in-memory locality groups are loaded lazily into the memory of the tablet server. Once loaded, column families that belong to such locality groups can be read without accessing the disk. This feature is useful for small pieces of data that are accessed frequently: we use it internally for the location column family in the `METADATA` table.

Compression

Clients can control whether or not the SSTables for a locality group are compressed, and if so, which compression format is used. The user-specified compression format is applied to each SSTable block (whose size is controllable via a locality group specific tuning parameter). Although we lose some space by compressing each block separately, we benefit in that small portions of an SSTable can be read without decompressing the entire file. Many clients use a two-pass custom compression scheme. The first pass uses Bentley and McIlroy's scheme [6], which compresses long common strings across a large window. The second pass uses a fast compression algorithm that looks for repetitions in a small 16 KB window of the data. Both compression passes are very fast—they encode at 100–200 MB/s, and decode at 400–1000 MB/s on modern machines.

Even though we emphasized speed instead of space reduction when choosing our compression algorithms, this two-pass compression scheme does surprisingly well. For example, in Webtable, we use this compression scheme to store Web page contents. In one experiment, we stored a large number of documents in a compressed locality group. For the purposes of the experiment, we limited ourselves to one version of each document instead of storing all versions available to us. The scheme achieved a 10-to-1 reduction in space. This is much better than typical Gzip reductions of 3-to-1 or 4-to-1 on HTML pages because of the way Webtable rows are laid out: all pages from a single host are stored close to each other. This allows the Bentley-McIlroy algorithm to identify large amounts of shared boilerplate in pages from the same host. Many applications, not just Webtable, choose their row names so that similar data ends up clustered, and therefore achieve very good compression ratios. Compression ratios get even better when we store multiple versions of the same value in Bigtable.

Caching for read performance

To improve read performance, tablet servers use two levels of caching. The Scan Cache is a higher-level cache that caches the key-value pairs returned by the SSTable interface to the tablet server code. The Block Cache is a lower-level cache that caches SSTables blocks that were read from GFS. The Scan Cache is most useful for applications that tend to read the same data repeatedly. The Block Cache is useful for applications that tend to read data that is close to the data they recently read (e.g., sequential reads, or random reads of different columns in the same locality group within a hot row).

Bloom filters

As described in Section 5.3, a read operation has to read from all SSTables that make up the state of a tablet. If these SSTables are not in memory, we may end up doing many disk accesses. We reduce the number of accesses by allowing clients to specify that Bloom filters [7] should be created for SSTables in a particular locality group. A Bloom filter allows us to ask whether an SSTable might contain any data for a specified row/column pair. For certain applications, a small amount of tablet server memory used for storing Bloom filters drastically reduces the number of disk seeks required for read operations. Our use of Bloom filters also implies that most lookups for non-existent rows or columns do not need to touch disk.

Commit-log implementation

If we kept the commit log for each tablet in a separate log file, a very large number of files would be written concurrently in GFS. Depending on the underlying file system implementation on each GFS server, these writes could cause a large number of disk seeks to write to the different physical log files. In addition, having separate log files per tablet also reduces the effectiveness of the group commit optimization, since groups would tend to be smaller. To fix these issues, we append mutations to a single commit log per tablet server, co-mingling mutations for different tablets in the same physical log file [18, 20].

Using one log provides significant performance benefits during normal operation, but it complicates recovery. When a tablet server dies, the tablets that it served will be moved to a large number of other tablet servers: each server typically loads a small number of the original server's tablets. To recover the state for a tablet, the new tablet server needs to reapply the mutations for that tablet from the commit log written by the original tablet server. However, the mutations for these tablets were co-mingled in the same physical log file. One approach would be for each new tablet server to read this full commit log file and apply just the entries needed for the tablets it needs to recover. However, under such a scheme, if 100 machines were each assigned a single tablet from a failed tablet server, then the log file would be read 100 times (once by each server).

We avoid duplicating log reads by first sorting the commit log entries in order of the keys $\langle\textsf{table}, \textsf{row name}, \textsf{log sequence number}\rangle$. In the sorted output, all mutations for a particular tablet are contiguous and can therefore be read efficiently with one disk seek followed by a sequential read. To parallelize the sorting, we partition the log file into 64 MB segments, and sort each segment in parallel on different tablet servers. This sorting process is coordinated by the master and is initiated when a tablet server indicates that it needs to recover mutations from some commit log file.

Writing commit logs to GFS sometimes causes performance hiccups for a variety of reasons (e.g., a GFS server machine involved in the write crashes, or the network paths traversed to reach the particular set of three GFS servers is suffering network congestion, or is heavily loaded). To protect mutations from GFS latency spikes, each tablet server actually has two log writing threads, each writing to its own log file; only one of these two threads is actively in use at a time. If writes to the active log file are performing poorly, the log file writing is switched to the other thread, and mutations that are in the commit log queue are written by the newly active log writing thread. Log entries contain sequence numbers to allow the recovery process to elide duplicated entries resulting from this log switching process.

Speeding up tablet recovery

If the master moves a tablet from one tablet server to another, the source tablet server first does a minor compaction on that tablet. This compaction reduces recovery time by reducing the amount of uncompacted state in the tablet server's commit log. After finishing this compaction, the tablet server stops serving the tablet. Before it actually unloads the tablet, the tablet server does another (usually very fast) minor compaction to eliminate any remaining uncompacted state in the tablet server's log that arrived while the first minor compaction was being performed. After this second minor compaction is complete, the tablet can be loaded on another tablet server without requiring any recovery of log entries.

Exploiting immutability

Besides the SSTable caches, various other parts of the Bigtable system have been simplified by the fact that all of the SSTables that we generate are immutable. For example, we do not need any synchronization of accesses to the file system when reading from SSTables. As a result, concurrency control over rows can be implemented very efficiently. The only mutable data structure that is accessed by both reads and writes is the memtable. To reduce contention during reads of the memtable, we make each memtable row copy-on-write and allow reads and writes to proceed in parallel.

Since SSTables are immutable, the problem of permanently removing deleted data is transformed to garbage collecting obsolete SSTables. Each tablet's SSTables are registered in the `METADATA` table. The master removes obsolete SSTables as a mark-and-sweep garbage collection [25] over the set of SSTables, where the `METADATA` table contains the set of roots.

Finally, the immutability of SSTables enables us to split tablets quickly. Instead of generating a new set of SSTables for each child tablet, we let the child tablets share the SSTables of the parent tablet.

7 Performance Evaluation

We set up a Bigtable cluster with N tablet servers to measure the performance and scalability of Bigtable as N is varied. The tablet servers were configured to use 1 GB of memory and to write to a GFS cell consisting of 1786 machines with two 400 GB IDE hard drives each. N client machines generated the Bigtable load used for these tests. (We used the same number of clients as tablet servers to ensure that clients were never a bottleneck.) Each machine had two dual-core Opteron 2 GHz chips, enough physical memory to hold the working set of all running processes, and a single gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.

The tablet servers and master, test clients, and GFS servers all ran on the same set of machines. Every machine ran a GFS server. Some of the machines also ran either a tablet server, or a client process, or processes from other jobs that were using the pool at the same time as these experiments.

R is the distinct number of Bigtable row keys involved in the test. R was chosen so that each benchmark read or wrote approximately 1 GB of data per tablet server.

The *sequential write* benchmark used row keys with names 0 to $R-1$. This space of row keys was partitioned into $10N$ equal-sized ranges. These ranges were assigned to the N clients by a central scheduler that as-

Experiment	# of Tablet Servers 1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

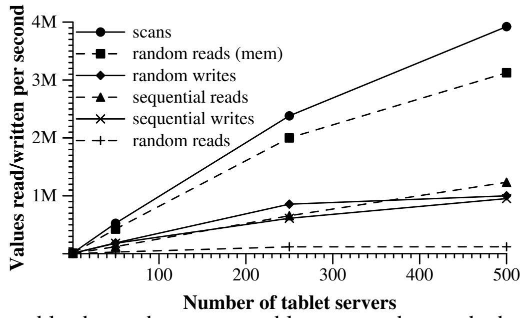


Figure 6: Number of 1000-byte values read/written per second. The table shows the rate per tablet server; the graph shows the aggregate rate.

signed the next available range to a client as soon as the client finished processing the previous range assigned to it. This dynamic assignment helped mitigate the effects of performance variations caused by other processes running on the client machines. We wrote a single string under each row key. Each string was generated randomly and was therefore uncompressible. In addition, strings under different row key were distinct, so no cross-row compression was possible. The *random write* benchmark was similar except that the row key was hashed modulo R immediately before writing so that the write load was spread roughly uniformly across the entire row space for the entire duration of the benchmark.

The *sequential read* benchmark generated row keys in exactly the same way as the sequential write benchmark, but instead of writing under the row key, it read the string stored under the row key (which was written by an earlier invocation of the sequential write benchmark). Similarly, the *random read* benchmark shadowed the operation of the random write benchmark.

The *scan* benchmark is similar to the sequential read benchmark, but uses support provided by the Bigtable API for scanning over all values in a row range. Using a scan reduces the number of RPCs executed by the benchmark since a single RPC fetches a large sequence of values from a tablet server.

The *random reads (mem)* benchmark is similar to the random read benchmark, but the locality group that contains the benchmark data is marked as *in-memory*, and therefore the reads are satisfied from the tablet server's memory instead of requiring a GFS read. For just this benchmark, we reduced the amount of data per tablet server from 1 GB to 100 MB so that it would fit comfortably in the memory available to the tablet server.

Figure 6 shows two views on the performance of our benchmarks when reading and writing 1000-byte values to Bigtable. The table shows the number of operations per second per tablet server; the graph shows the aggregate number of operations per second.

Single tablet-server performance

Let us first consider performance with just one tablet server. Random reads are slower than all other operations by an order of magnitude or more. Each random read involves the transfer of a 64 KB SSTable block over the network from GFS to a tablet server, out of which only a single 1000-byte value is used. The tablet server executes approximately 1200 reads per second, which translates into approximately 75 MB/s of data read from GFS. This bandwidth is enough to saturate the tablet server CPUs because of overheads in our networking stack, SSTable parsing, and Bigtable code, and is also almost enough to saturate the network links used in our system. Most Bigtable applications with this type of an access pattern reduce the block size to a smaller value, typically 8KB.

Random reads from memory are much faster since each 1000-byte read is satisfied from the tablet server's local memory without fetching a large 64 KB block from GFS.

Random and sequential writes perform better than random reads since each tablet server appends all incoming writes to a single commit log and uses group commit to stream these writes efficiently to GFS. There is no significant difference between the performance of random writes and sequential writes; in both cases, all writes to the tablet server are recorded in the same commit log.

Sequential reads perform better than random reads since every 64 KB SSTable block that is fetched from GFS is stored into our block cache, where it is used to serve the next 64 read requests.

Scans are even faster since the tablet server can return a large number of values in response to a single client RPC, and therefore RPC overhead is amortized over a large number of values.

Scaling

Aggregate throughput increases dramatically, by over a factor of a hundred, as we increase the number of tablet servers in the system from 1 to 500. For example, the

# of tablet servers	# of clusters
0 .. 19	259
20 .. 49	47
50 .. 99	20
100 .. 499	50
> 500	12

Table 1: Distribution of number of tablet servers in Bigtable clusters.

performance of random reads from memory increases by almost a factor of 300 as the number of tablet server increases by a factor of 500. This behavior occurs because the bottleneck on performance for this benchmark is the individual tablet server CPU.

However, performance does not increase linearly. For most benchmarks, there is a significant drop in per-server throughput when going from 1 to 50 tablet servers. This drop is caused by imbalance in load in multiple server configurations, often due to other processes contending for CPU and network. Our load balancing algorithm attempts to deal with this imbalance, but cannot do a perfect job for two main reasons: rebalancing is throttled to reduce the number of tablet movements (a tablet is unavailable for a short time, typically less than one second, when it is moved), and the load generated by our benchmarks shifts around as the benchmark progresses.

The random read benchmark shows the worst scaling (an increase in aggregate throughput by only a factor of 100 for a 500-fold increase in number of servers). This behavior occurs because (as explained above) we transfer one large 64KB block over the network for every 1000-byte read. This transfer saturates various shared 1 Gigabit links in our network and as a result, the per-server throughput drops significantly as we increase the number of machines.

8 Real Applications

As of August 2006, there are 388 non-test Bigtable clusters running in various Google machine clusters, with a combined total of about 24,500 tablet servers. Table 1 shows a rough distribution of tablet servers per cluster. Many of these clusters are used for development purposes and therefore are idle for significant periods. One group of 14 busy clusters with 8069 total tablet servers saw an aggregate volume of more than 1.2 million requests per second, with incoming RPC traffic of about 741 MB/s and outgoing RPC traffic of about 16 GB/s.

Table 2 provides some data about a few of the tables currently in use. Some tables store data that is served to users, whereas others store data for batch processing; the tables range widely in total size, average cell size, percentage of data served from memory, and complexity of the table schema. In the rest of this section, we briefly describe how three product teams use Bigtable.

8.1 Google Analytics

Google Analytics (analytics.google.com) is a service that helps webmasters analyze traffic patterns at their web sites. It provides aggregate statistics, such as the number of unique visitors per day and the page views per URL per day, as well as site-tracking reports, such as the percentage of users that made a purchase, given that they earlier viewed a specific page.

To enable the service, webmasters embed a small JavaScript program in their web pages. This program is invoked whenever a page is visited. It records various information about the request in Google Analytics, such as a user identifier and information about the page being fetched. Google Analytics summarizes this data and makes it available to webmasters.

We briefly describe two of the tables used by Google Analytics. The raw click table (˜200 TB) maintains a row for each end-user session. The row name is a tuple containing the website's name and the time at which the session was created. This schema ensures that sessions that visit the same web site are contiguous, and that they are sorted chronologically. This table compresses to 14% of its original size.

The summary table (˜20 TB) contains various predefined summaries for each website. This table is generated from the raw click table by periodically scheduled MapReduce jobs. Each MapReduce job extracts recent session data from the raw click table. The overall system's throughput is limited by the throughput of GFS. This table compresses to 29% of its original size.

8.2 Google Earth

Google operates a collection of services that provide users with access to high-resolution satellite imagery of the world's surface, both through the web-based Google Maps interface (maps.google.com) and through the Google Earth (earth.google.com) custom client software. These products allow users to navigate across the world's surface: they can pan, view, and annotate satellite imagery at many different levels of resolution. This system uses one table to preprocess data, and a different set of tables for serving client data.

The preprocessing pipeline uses one table to store raw imagery. During preprocessing, the imagery is cleaned and consolidated into final serving data. This table contains approximately 70 terabytes of data and therefore is served from disk. The images are efficiently compressed already, so Bigtable compression is disabled.

Project name	**Table size (TB)**	**Compression ratio**	**# Cells (billions)**	**# Column Families**	**# Locality Groups**	**% in memory**	**Latency-sensitive?**
Crawl	800	11%	1000	16	8	0%	No
Crawl	50	33%	200	2	2	0%	No
Google Analytics	20	29%	10	1	1	0%	Yes
Google Analytics	200	14%	80	1	1	0%	Yes
Google Base	2	31%	10	29	3	15%	Yes
Google Earth	0.5	64%	8	7	2	33%	Yes
Google Earth	70	–	9	8	3	0%	No
Orkut	9	–	0.9	8	5	1%	Yes
Personalized Search	4	47%	6	93	11	5%	Yes

Table 2: Characteristics of a few tables in production use. *Table size* (measured before compression) and *# Cells* indicate approximate sizes. *Compression ratio* is not given for tables that have compression disabled.

Each row in the imagery table corresponds to a single geographic segment. Rows are named to ensure that adjacent geographic segments are stored near each other. The table contains a column family to keep track of the sources of data for each segment. This column family has a large number of columns: essentially one for each raw data image. Since each segment is only built from a few images, this column family is very sparse.

The preprocessing pipeline relies heavily on MapReduce over Bigtable to transform data. The overall system processes over 1 MB/sec of data per tablet server during some of these MapReduce jobs.

The serving system uses one table to index data stored in GFS. This table is relatively small (˜500 GB), but it must serve tens of thousands of queries per second per datacenter with low latency. As a result, this table is hosted across hundreds of tablet servers and contains in-memory column families.

8.3 Personalized Search

Personalized Search (www.google.com/psearch) is an opt-in service that records user queries and clicks across a variety of Google properties such as web search, images, and news. Users can browse their search histories to revisit their old queries and clicks, and they can ask for personalized search results based on their historical Google usage patterns.

Personalized Search stores each user's data in Bigtable. Each user has a unique userid and is assigned a row named by that userid. All user actions are stored in a table. A separate column family is reserved for each type of action (for example, there is a column family that stores all web queries). Each data element uses as its Bigtable timestamp the time at which the corresponding user action occurred. Personalized Search generates user profiles using a MapReduce over Bigtable. These user profiles are used to personalize live search results.

The Personalized Search data is replicated across several Bigtable clusters to increase availability and to reduce latency due to distance from clients. The Personalized Search team originally built a client-side replication mechanism on top of Bigtable that ensured eventual consistency of all replicas. The current system now uses a replication subsystem that is built into the servers.

The design of the Personalized Search storage system allows other groups to add new per-user information in their own columns, and the system is now used by many other Google properties that need to store per-user configuration options and settings. Sharing a table amongst many groups resulted in an unusually large number of column families. To help support sharing, we added a simple quota mechanism to Bigtable to limit the storage consumption by any particular client in shared tables; this mechanism provides some isolation between the various product groups using this system for per-user information storage.

9 Lessons

In the process of designing, implementing, maintaining, and supporting Bigtable, we gained useful experience and learned several interesting lessons.

One lesson we learned is that large distributed systems are vulnerable to many types of failures, not just the standard network partitions and fail-stop failures assumed in many distributed protocols. For example, we have seen problems due to all of the following causes: memory and network corruption, large clock skew, hung machines, extended and asymmetric network partitions, bugs in other systems that we are using (Chubby for example), overflow of GFS quotas, and planned and unplanned hardware maintenance. As we have gained more experience with these problems, we have addressed them by changing various protocols. For example, we added checksumming to our RPC mechanism. We also handled some problems by removing assumptions made by one part of the system about another part. For example, we stopped assuming a given Chubby operation could return only one of a fixed set of errors.

Another lesson we learned is that it is important to delay adding new features until it is clear how the new features will be used. For example, we initially planned to support general-purpose transactions in our API. Because we did not have an immediate use for them, however, we did not implement them. Now that we have many real applications running on Bigtable, we have been able to examine their actual needs, and have discovered that most applications require only single-row transactions. Where people have requested distributed transactions, the most important use is for maintaining secondary indices, and we plan to add a specialized mechanism to satisfy this need. The new mechanism will be less general than distributed transactions, but will be more efficient (especially for updates that span hundreds of rows or more) and will also interact better with our scheme for optimistic cross-data-center replication.

A practical lesson that we learned from supporting Bigtable is the importance of proper system-level monitoring (i.e., monitoring both Bigtable itself, as well as the client processes using Bigtable). For example, we extended our RPC system so that for a sample of the RPCs, it keeps a detailed trace of the important actions done on behalf of that RPC. This feature has allowed us to detect and fix many problems such as lock contention on tablet data structures, slow writes to GFS while committing Bigtable mutations, and stuck accesses to the `METADATA` table when `METADATA` tablets are unavailable. Another example of useful monitoring is that every Bigtable cluster is registered in Chubby. This allows us to track down all clusters, discover how big they are, see which versions of our software they are running, how much traffic they are receiving, and whether or not there are any problems such as unexpectedly large latencies.

The most important lesson we learned is the value of simple designs. Given both the size of our system (about 100,000 lines of non-test code), as well as the fact that code evolves over time in unexpected ways, we have found that code and design clarity are of immense help in code maintenance and debugging. One example of this is our tablet-server membership protocol. Our first protocol was simple: the master periodically issued leases to tablet servers, and tablet servers killed themselves if their lease expired. Unfortunately, this protocol reduced availability significantly in the presence of network problems, and was also sensitive to master recovery time. We redesigned the protocol several times until we had a protocol that performed well. However, the resulting protocol was too complex and depended on the behavior of Chubby features that were seldom exercised by other applications. We discovered that we were spending an inordinate amount of time debugging obscure corner cases, not only in Bigtable code, but also in Chubby code. Eventually, we scrapped this protocol and moved to a newer simpler protocol that depends solely on widely-used Chubby features.

10 Related Work

The Boxwood project [24] has components that overlap in some ways with Chubby, GFS, and Bigtable, since it provides for distributed agreement, locking, distributed chunk storage, and distributed B-tree storage. In each case where there is overlap, it appears that the Boxwood's component is targeted at a somewhat lower level than the corresponding Google service. The Boxwood project's goal is to provide infrastructure for building higher-level services such as file systems or databases, while the goal of Bigtable is to directly support client applications that wish to store data.

Many recent projects have tackled the problem of providing distributed storage or higher-level services over wide area networks, often at "Internet scale." This includes work on distributed hash tables that began with projects such as CAN [29], Chord [32], Tapestry [37], and Pastry [30]. These systems address concerns that do not arise for Bigtable, such as highly variable bandwidth, untrusted participants, or frequent reconfiguration; decentralized control and Byzantine fault tolerance are not Bigtable goals.

In terms of the distributed data storage model that one might provide to application developers, we believe the key-value pair model provided by distributed B-trees or distributed hash tables is too limiting. Key-value pairs are a useful building block, but they should not be the only building block one provides to developers. The model we chose is richer than simple key-value pairs, and supports sparse semi-structured data. Nonetheless, it is still simple enough that it lends itself to a very efficient flat-file representation, and it is transparent enough (via locality groups) to allow our users to tune important behaviors of the system.

Several database vendors have developed parallel databases that can store large volumes of data. Oracle's Real Application Cluster database [27] uses shared disks to store data (Bigtable uses GFS) and a distributed lock manager (Bigtable uses Chubby). IBM's DB2 Parallel Edition [4] is based on a shared-nothing [33] architecture similar to Bigtable. Each DB2 server is responsible for a subset of the rows in a table which it stores in a local relational database. Both products provide a complete relational model with transactions.

Bigtable locality groups realize similar compression and disk read performance benefits observed for other systems that organize data on disk using column-based rather than row-based storage, including C-Store [1, 34] and commercial products such as Sybase IQ [15, 36], SenSage [31], KDB+ [22], and the ColumnBM storage layer in MonetDB/X100 [38]. Another system that does vertical and horizontal data partioning into flat files and achieves good data compression ratios is AT&T's Daytona database [19]. Locality groups do not support CPU-cache-level optimizations, such as those described by Ailamaki [2].

The manner in which Bigtable uses memtables and SSTables to store updates to tablets is analogous to the way that the Log-Structured Merge Tree [26] stores updates to index data. In both systems, sorted data is buffered in memory before being written to disk, and reads must merge data from memory and disk.

C-Store and Bigtable share many characteristics: both systems use a shared-nothing architecture and have two different data structures, one for recent writes, and one for storing long-lived data, with a mechanism for moving data from one form to the other. The systems differ significantly in their API: C-Store behaves like a relational database, whereas Bigtable provides a lower level read and write interface and is designed to support many thousands of such operations per second per server. C-Store is also a "read-optimized relational DBMS", whereas Bigtable provides good performance on both read-intensive and write-intensive applications.

Bigtable's load balancer has to solve some of the same kinds of load and memory balancing problems faced by shared-nothing databases (e.g., [11, 35]). Our problem is somewhat simpler: (1) we do not consider the possibility of multiple copies of the same data, possibly in alternate forms due to views or indices; (2) we let the user tell us what data belongs in memory and what data should stay on disk, rather than trying to determine this dynamically; (3) we have no complex queries to execute or optimize.

11 Conclusions

We have described Bigtable, a distributed system for storing structured data at Google. Bigtable clusters have been in production use since April 2005, and we spent roughly seven person-years on design and implementation before that date. As of August 2006, more than sixty projects are using Bigtable. Our users like the performance and high availability provided by the Bigtable implementation, and that they can scale the capacity of their clusters by simply adding more machines to the system as their resource demands change over time.

Given the unusual interface to Bigtable, an interesting question is how difficult it has been for our users to adapt to using it. New users are sometimes uncertain of how to best use the Bigtable interface, particularly if they are accustomed to using relational databases that support general-purpose transactions. Nevertheless, the fact that many Google products successfully use Bigtable demonstrates that our design works well in practice.

We are in the process of implementing several additional Bigtable features, such as support for secondary indices and infrastructure for building cross-data-center replicated Bigtables with multiple master replicas. We have also begun deploying Bigtable as a service to product groups, so that individual groups do not need to maintain their own clusters. As our service clusters scale, we will need to deal with more resource-sharing issues within Bigtable itself [3, 5].

Finally, we have found that there are significant advantages to building our own storage solution at Google. We have gotten a substantial amount of flexibility from designing our own data model for Bigtable. In addition, our control over Bigtable's implementation, and the other Google infrastructure upon which Bigtable depends, means that we can remove bottlenecks and inefficiencies as they arise.

Acknowledgements

We thank the anonymous reviewers, David Nagle, and our shepherd Brad Calder, for their feedback on this paper. The Bigtable system has benefited greatly from the feedback of our many users within Google. In addition, we thank the following people for their contributions to Bigtable: Dan Aguayo, Sameer Ajmani, Zhifeng Chen, Bill Coughran, Mike Epstein, Healfdene Goguen, Robert Griesemer, Jeremy Hylton, Josh Hyman, Alex Khesin, Joanna Kulik, Alberto Lerner, Sherry Listgarten, Mike Maloney, Eduardo Pinheiro, Kathy Polizzi, Frank Yellin, and Arthur Zwiegincew.

References

[1] Abadi, D. J., Madden, S. R., and Ferreira, M. C. Integrating compression and execution in column-oriented database systems. *Proc. of SIGMOD* (2006).

[2] Ailamaki, A., DeWitt, D. J., Hill, M. D., and Skounakis, M. Weaving relations for cache performance. In *The VLDB Journal* (2001), pp. 169–180.

[3] Banga, G., Druschel, P., and Mogul, J. C. Resource containers: A new facility for resource management in server systems. In *Proc. of the 3rd OSDI* (Feb. 1999), pp. 45–58.

[4] Baru, C. K., Fecteau, G., Goyal, A., Hsiao, H., Jhingran, A., Padmanabhan, S., Copeland,

G. P., AND WILSON, W. G. DB2 parallel edition. *IBM Systems Journal 34*, 2 (1995), 292–322.

[5] BAVIER, A., BOWMAN, M., CHUN, B., CULLER, D., KARLIN, S., PETERSON, L., ROSCOE, T., SPALINK, T., AND WAWRZONIAK, M. Operating system support for planetary-scale network services. In *Proc. of the 1st NSDI* (Mar. 2004), pp. 253–266.

[6] BENTLEY, J. L., AND MCILROY, M. D. Data compression using long common strings. In *Data Compression Conference* (1999), pp. 287–295.

[7] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *CACM 13*, 7 (1970), 422–426.

[8] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of the 7th OSDI* (Nov. 2006).

[9] CHANDRA, T., GRIESEMER, R., AND REDSTONE, J. Paxos made live — An engineering perspective. In *Proc. of PODC* (2007).

[10] COMER, D. Ubiquitous B-tree. *Computing Surveys 11*, 2 (June 1979), 121–137.

[11] COPELAND, G. P., ALEXANDER, W., BOUGHTER, E. E., AND KELLER, T. W. Data placement in Bubba. In *Proc. of SIGMOD* (1988), pp. 99–108.

[12] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *Proc. of the 6th OSDI* (Dec. 2004), pp. 137–150.

[13] DEWITT, D., KATZ, R., OLKEN, F., SHAPIRO, L., STONEBRAKER, M., AND WOOD, D. Implementation techniques for main memory database systems. In *Proc. of SIGMOD* (June 1984), pp. 1–8.

[14] DEWITT, D. J., AND GRAY, J. Parallel database systems: The future of high performance database systems. *CACM 35*, 6 (June 1992), 85–98.

[15] FRENCH, C. D. One size fits all database architectures do not work for DSS. In *Proc. of SIGMOD* (May 1995), pp. 449–450.

[16] GAWLICK, D., AND KINKADE, D. Varieties of concurrency control in IMS/VS fast path. *Database Engineering Bulletin 8*, 2 (1985), 3–10.

[17] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proc. of the 19th ACM SOSP* (Dec. 2003), pp. 29–43.

[18] GRAY, J. Notes on database operating systems. In *Operating Systems — An Advanced Course*, vol. 60 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[19] GREER, R. Daytona and the fourth-generation language Cymbal. In *Proc. of SIGMOD* (1999), pp. 525–526.

[20] HAGMANN, R. Reimplementing the Cedar file system using logging and group commit. In *Proc. of the 11th SOSP* (Dec. 1987), pp. 155–162.

[21] HARTMAN, J. H., AND OUSTERHOUT, J. K. The Zebra striped network file system. In *Proc. of the 14th SOSP* (Asheville, NC, 1993), pp. 29–43.

[22] KX.COM. kx.com/products/database.php. Product page.

[23] LAMPORT, L. The part-time parliament. *ACM TOCS 16*, 2 (1998), 133–169.

[24] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. of the 6th OSDI* (Dec. 2004), pp. 105–120.

[25] MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine. *CACM 3*, 4 (Apr. 1960), 184–195.

[26] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (LSM-tree). *Acta Inf. 33*, 4 (1996), 351–385.

[27] ORACLE.COM. www.oracle.com/technology/products/-database/clustering/index.html. Product page.

[28] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming Journal 13*, 4 (2005), 227–298.

[29] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In *Proc. of SIGCOMM* (Aug. 2001), pp. 161–172.

[30] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of Middleware 2001* (Nov. 2001), pp. 329–350.

[31] SENSAGE.COM. sensage.com/products-sensage.htm. Product page.

[32] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of SIGCOMM* (Aug. 2001), pp. 149–160.

[33] STONEBRAKER, M. The case for shared nothing. *Database Engineering Bulletin 9*, 1 (Mar. 1986), 4–9.

[34] STONEBRAKER, M., ABADI, D. J., BATKIN, A., CHEN, X., CHERNIACK, M., FERREIRA, M., LAU, E., LIN, A., MADDEN, S., O'NEIL, E., O'NEIL, P., RASIN, A., TRAN, N., AND ZDONIK, S. C-Store: A column-oriented DBMS. In *Proc. of VLDB* (Aug. 2005), pp. 553–564.

[35] STONEBRAKER, M., AOKI, P. M., DEVINE, R., LITWIN, W., AND OLSON, M. A. Mariposa: A new architecture for distributed data. In *Proc. of the Tenth ICDE* (1994), IEEE Computer Society, pp. 54–65.

[36] SYBASE.COM. www.sybase.com/products/database-servers/sybaseiq. Product page.

[37] ZHAO, B. Y., KUBIATOWICZ, J., AND JOSEPH, A. D. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, CS Division, UC Berkeley, Apr. 2001.

[38] ZUKOWSKI, M., BONCZ, P. A., NES, N., AND HEMAN, S. MonetDB/X100 — A DBMS in the CPU cache. *IEEE Data Eng. Bull. 28*, 2 (2005), 17–22.

The Chubby lock service for loosely-coupled distributed systems

Mike Burrows, *Google Inc.*

Abstract

We describe our experiences with the Chubby lock service, which is intended to provide coarse-grained locking as well as reliable (though low-volume) storage for a loosely-coupled distributed system. Chubby provides an interface much like a distributed file system with advisory locks, but the design emphasis is on availability and reliability, as opposed to high performance. Many instances of the service have been used for over a year, with several of them each handling a few tens of thousands of clients concurrently. The paper describes the initial design and expected use, compares it with actual use, and explains how the design had to be modified to accommodate the differences.

1 Introduction

This paper describes a *lock service* called Chubby. It is intended for use within a loosely-coupled distributed system consisting of moderately large numbers of small machines connected by a high-speed network. For example, a Chubby instance (also known as a Chubby *cell*) might serve ten thousand 4-processor machines connected by 1Gbit/s Ethernet. Most Chubby cells are confined to a single data centre or machine room, though we do run at least one Chubby cell whose replicas are separated by thousands of kilometres.

The purpose of the lock service is to allow its clients to synchronize their activities and to agree on basic information about their environment. The primary goals included reliability, availability to a moderately large set of clients, and easy-to-understand semantics; throughput and storage capacity were considered secondary. Chubby's client interface is similar to that of a simple file system that performs *whole-file* reads and writes, augmented with advisory locks and with notification of various events such as file modification.

We expected Chubby to help developers deal with coarse-grained synchronization within their systems, and in particular to deal with the problem of electing a leader from among a set of otherwise equivalent servers. For example, the Google File System [7] uses a Chubby lock to appoint a GFS master server, and Bigtable [3] uses Chubby in several ways: to elect a master, to allow the master to discover the servers it controls, and to permit clients to find the master. In addition, both GFS and Bigtable use Chubby as a well-known and available location to store a small amount of meta-data; in effect they use Chubby as the root of their distributed data structures. Some services use locks to partition work (at a coarse grain) between several servers.

Before Chubby was deployed, most distributed systems at Google used *ad hoc* methods for primary election (when work could be duplicated without harm), or required operator intervention (when correctness was essential). In the former case, Chubby allowed a small saving in computing effort. In the latter case, it achieved a significant improvement in availability in systems that no longer required human intervention on failure.

Readers familiar with distributed computing will recognize the election of a primary among peers as an instance of the *distributed consensus* problem, and realize we require a solution using *asynchronous* communication; this term describes the behaviour of the vast majority of real networks, such as Ethernet or the Internet, which allow packets to be lost, delayed, and reordered. (Practitioners should normally beware of protocols based on models that make stronger assumptions on the environment.) Asynchronous consensus is solved by the *Paxos* protocol [12, 13]. The same protocol was used by Oki and Liskov (see their paper on *viewstamped replication* [19, §4]), an equivalence noted by others [14, §6]. Indeed, all working protocols for asynchronous consensus we have so far encountered have Paxos at their core. Paxos maintains safety without timing assumptions, but clocks must be introduced to ensure liveness; this overcomes the impossibility result of Fischer *et al.* [5, §1].

Building Chubby was an engineering effort required to fill the needs mentioned above; it was not research. We claim no new algorithms or techniques. The purpose of this paper is to describe what we did and why, rather than to advocate it. In the sections that follow, we describe Chubby's design and implementation, and how it has changed in the light of experience. We describe unexpected ways in which Chubby has been used, and features that proved to be mistakes. We omit details that are covered elsewhere in the literature, such as the details of a consensus protocol or an RPC system.

2 Design

2.1 Rationale

One might argue that we should have built a library embodying Paxos, rather than a library that accesses a centralized lock service, even a highly reliable one. A client Paxos library would depend on *no* other servers (besides the name service), and would provide a standard framework for programmers, assuming their services can be implemented as state machines. Indeed, we provide such a client library that is independent of Chubby.

Nevertheless, a lock service has some advantages over a client library. First, our developers sometimes do not plan for high availability in the way one would wish. Often their systems start as prototypes with little load and loose availability guarantees; invariably the code has not been specially structured for use with a consensus protocol. As the service matures and gains clients, availability becomes more important; replication and primary election are then added to an existing design. While this could be done with a library that provides distributed consensus, a lock server makes it easier to maintain existing program structure and communication patterns. For example, to elect a master which then writes to an existing file server requires adding just two statements and one RPC parameter to an existing system: One would acquire a lock to become master, pass an additional integer (the lock acquisition count) with the write RPC, and add an if-statement to the file server to reject the write if the acquisition count is lower than the current value (to guard against delayed packets). We have found this technique easier than making existing servers participate in a consensus protocol, and especially so if compatibility must be maintained during a transition period.

Second, many of our services that elect a primary or that partition data between their components need a mechanism for advertising the results. This suggests that we should allow clients to store and fetch small quantities of data—that is, to read and write small files. This could be done with a name service, but our experience has been that the lock service itself is well-suited for this task, both because this reduces the number of servers on which a client depends, and because the consistency features of the protocol are shared. Chubby's success as a name server owes much to its use of consistent client caching, rather than time-based caching. In particular, we found that developers greatly appreciated not having to choose a cache timeout such as the DNS time-to-live value, which if chosen poorly can lead to high DNS load, or long client fail-over times.

Third, a lock-based interface is more familiar to our programmers. Both the replicated state machine of Paxos and the critical sections associated with exclusive locks can provide the programmer with the illusion of sequential programming. However, many programmers have come across locks before, and think they know to use them. Ironically, such programmers are usually wrong, especially when they use locks in a distributed system; few consider the effects of independent machine failures on locks in a system with asynchronous communications. Nevertheless, the apparent familiarity of locks overcomes a hurdle in persuading programmers to use a reliable mechanism for distributed decision making.

Last, distributed-consensus algorithms use quorums to make decisions, so they use several replicas to achieve high availability. For example, Chubby itself usually has five replicas in each cell, of which three must be running for the cell to be up. In contrast, if a client system uses a lock service, even a single client can obtain a lock and make progress safely. Thus, a lock service reduces the number of servers needed for a reliable client system to make progress. In a loose sense, one can view the lock service as a way of providing a generic electorate that allows a client system to make decisions correctly when less than a majority of its own members are up. One might imagine solving this last problem in a different way: by providing a "consensus service", using a number of servers to provide the "acceptors" in the Paxos protocol. Like a lock service, a consensus service would allow clients to make progress safely even with only one active client process; a similar technique has been used to reduce the number of state machines needed for Byzantine fault tolerance [24]. However, assuming a consensus service is not used exclusively to provide locks (which reduces it to a lock service), this approach solves none of the other problems described above.

These arguments suggest two key design decisions:

- We chose a lock service, as opposed to a library or service for consensus, and
- we chose to serve small-files to permit elected primaries to advertise themselves and their parameters, rather than build and maintain a second service.

Some decisions follow from our expected use and from our environment:

- A service advertising its primary via a Chubby file may have thousands of clients. Therefore, we must allow thousands of clients to observe this file, preferably without needing many servers.
- Clients and replicas of a replicated service may wish to know when the service's primary changes. This

suggests that an event notification mechanism would be useful to avoid polling.

- Even if clients need not poll files periodically, many will; this is a consequence of supporting many developers. Thus, caching of files is desirable.
- Our developers are confused by non-intuitive caching semantics, so we prefer consistent caching.
- To avoid both financial loss and jail time, we provide security mechanisms, including access control.

A choice that may surprise some readers is that we do not expect lock use to be *fine-grained*, in which they might be held only for a short duration (seconds or less); instead, we expect *coarse-grained* use. For example, an application might use a lock to elect a primary, which would then handle all access to that data for a considerable time, perhaps hours or days. These two styles of use suggest different requirements from a lock server.

Coarse-grained locks impose far less load on the lock server. In particular, the lock-acquisition rate is usually only weakly related to the transaction rate of the client applications. Coarse-grained locks are acquired only rarely, so temporary lock server unavailability delays clients less. On the other hand, the transfer of a lock from client to client may require costly recovery procedures, so one would not wish a fail-over of a lock server to cause locks to be lost. Thus, it is good for coarse-grained locks to survive lock server failures, there is little concern about the overhead of doing so, and such locks allow many clients to be adequately served by a modest number of lock servers with somewhat lower availability.

Fine-grained locks lead to different conclusions. Even brief unavailability of the lock server may cause many clients to stall. Performance and the ability to add new servers at will are of great concern because the transaction rate at the lock service grows with the combined transaction rate of clients. It can be advantageous to reduce the overhead of locking by not maintaining locks across lock server failure, and the time penalty for dropping locks every so often is not severe because locks are held for short periods. (Clients must be prepared to lose locks during network partitions, so the loss of locks on lock server fail-over introduces no new recovery paths.)

Chubby is intended to provide only coarse-grained locking. Fortunately, it is straightforward for clients to implement their own fine-grained locks tailored to their application. An application might partition its locks into groups and use Chubby's coarse-grained locks to allocate these lock groups to application-specific lock servers. Little state is needed to maintain these fine-grain locks; the servers need only keep a non-volatile, monotonically-increasing acquisition counter that is rarely updated. Clients can learn of lost locks at unlock time, and if a simple fixed-length lease is used, the protocol can be simple and efficient. The most important benefits of this scheme are that our client developers become responsible for the provisioning of the servers needed to support their load, yet are relieved of the complexity of implementing consensus themselves.

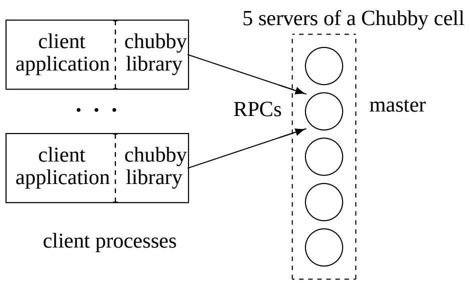


Figure 1: System structure

2.2 System structure

Chubby has two main components that communicate via RPC: a server, and a library that client applications link against; see Figure 1. All communication between Chubby clients and the servers is mediated by the client library. An optional third component, a proxy server, is discussed in Section 3.1.

A Chubby cell consists of a small set of servers (typically five) known as *replicas*, placed so as to reduce the likelihood of correlated failure (for example, in different racks). The replicas use a distributed consensus protocol to elect a *master*; the master must obtain votes from a majority of the replicas, plus promises that those replicas will not elect a different master for an interval of a few seconds known as the *master lease*. The master lease is periodically renewed by the replicas provided the master continues to win a majority of the vote.

The replicas maintain copies of a simple database, but only the master initiates reads and writes of this database. All other replicas simply copy updates from the master, sent using the consensus protocol.

Clients find the master by sending master location requests to the replicas listed in the DNS. Non-master replicas respond to such requests by returning the identity of the master. Once a client has located the master, the client directs all requests to it either until it ceases to respond, or until it indicates that it is no longer the master. Write requests are propagated via the consensus protocol to all replicas; such requests are acknowledged when the write has reached a majority of the replicas in the cell. Read requests are satisfied by the master alone; this is safe provided the master lease has not expired, as no other master can possibly exist. If a master fails, the other replicas run the election protocol when their master leases expire; a new master will typically be elected in a few seconds. For example, two recent elections took 6s and 4s, but we see values as high as 30s (§4.1).

If a replica fails and does not recover for a few hours, a simple replacement system selects a fresh machine from a free pool and starts the lock server binary on it. It then updates the DNS tables, replacing the IP address of the failed replica with that of the new one. The current master polls the DNS periodically and eventually notices the change. It then updates the list of the cell's members in the cell's database; this list is kept consistent across all the members via the normal replication protocol. In the meantime, the new replica obtains a recent copy of the database from a combination of backups stored on file servers and updates from active replicas. Once the new replica has processed a request that the current master is waiting to commit, the replica is permitted to vote in the elections for new master.

2.3 Files, directories, and handles

Chubby exports a file system interface similar to, but simpler than that of UNIX [22]. It consists of a strict tree of files and directories in the usual way, with name components separated by slashes. A typical name is:

```
/ls/foo/wombat/pouch
```

The `ls` prefix is common to all Chubby names, and stands for *lock service*. The second component (`foo`) is the name of a Chubby cell; it is resolved to one or more Chubby servers via DNS lookup. A special cell name `local` indicates that the client's local Chubby cell should be used; this is usually one in the same building and thus the one most likely to be accessible. The remainder of the name, `/wombat/pouch`, is interpreted within the named Chubby cell. Again following UNIX, each directory contains a list of child files and directories, while each file contains a sequence of uninterpreted bytes.

Because Chubby's naming structure resembles a file system, we were able to make it available to applications both with its own specialized API, and via interfaces used by our other file systems, such as the Google File System. This significantly reduced the effort needed to write basic browsing and name space manipulation tools, and reduced the need to educate casual Chubby users.

The design differs from UNIX in a ways that ease distribution. To allow the files in different directories to be served from different Chubby masters, we do not expose operations that can move files from one directory to another, we do not maintain directory modified times, and we avoid path-dependent permission semantics (that is, access to a file is controlled by the permissions on the file itself rather than on directories on the path leading to the file). To make it easier to cache file meta-data, the system does not reveal last-access times.

The name space contains only files and directories, collectively called *nodes*. Every such node has only one name within its cell; there are no symbolic or hard links.

Nodes may be either permanent or ephemeral. Any node may be deleted explicitly, but ephemeral nodes are also deleted if no client has them open (and, for directories, they are empty). Ephemeral files are used as temporary files, and as indicators to others that a client is alive. Any node can act as an advisory reader/writer lock; these locks are described in more detail in Section 2.4.

Each node has various meta-data, including three names of access control lists (ACLs) used to control reading, writing and changing the ACL names for the node. Unless overridden, a node inherits the ACL names of its parent directory on creation. ACLs are themselves files located in an ACL directory, which is a well-known part of the cell's local name space. These ACL files consist of simple lists of names of principals; readers may be reminded of Plan 9's *groups* [21]. Thus, if file F's write ACL name is `foo`, and the ACL directory contains a file `foo` that contains an entry `bar`, then user `bar` is permitted to write F. Users are authenticated by a mechanism built into the RPC system. Because Chubby's ACLs are simply files, they are automatically available to other services that wish to use similar access control mechanisms.

The per-node meta-data includes four monotonically-increasing 64-bit numbers that allow clients to detect changes easily:

- an instance number; greater than the instance number of any previous node with the same name.
- a content generation number (files only); this increases when the file's contents are written.
- a lock generation number; this increases when the node's lock transitions from *free* to *held*.
- an ACL generation number; this increases when the node's ACL names are written.

Chubby also exposes a 64-bit file-content checksum so clients may tell whether files differ.

Clients open nodes to obtain *handles* that are analogous to UNIX file descriptors. Handles include:

- check digits that prevent clients from creating or guessing handles, so full access control checks need be performed only when handles are created (compare with UNIX, which checks its permissions bits at open time, but not at each read/write because file descriptors cannot be forged).
- a sequence number that allows a master to tell whether a handle was generated by it or by a previous master.
- mode information provided at open time to allow the master to recreate its state if an old handle is presented to a newly restarted master.

2.4 Locks and sequencers

Each Chubby file and directory can act as a reader-writer lock: either one client handle may hold the lock in exclusive (writer) mode, or any number of client handles may hold the lock in shared (reader) mode. Like the mutexes known to most programmers, locks are *advisory*. That is, they conflict only with other attempts to acquire the same lock: holding a lock called F neither is necessary to access the file F, nor prevents other clients from doing so. We rejected *mandatory* locks, which make locked objects inaccessible to clients not holding their locks:

- Chubby locks often protect resources implemented by other services, rather than just the file associated with the lock. To enforce mandatory locking in a meaningful way would have required us to make more extensive modification of these services.
- We did not wish to force users to shut down applications when they needed to access locked files for debugging or administrative purposes. In a complex system, it is harder to use the approach employed on most personal computers, where administrative software can break mandatory locks simply by instructing the user to shut down his applications or to reboot.
- Our developers perform error checking in the conventional way, by writing assertions such as "lock X is held", so they benefit little from mandatory checks. Buggy or malicious processes have many opportunities to corrupt data when locks are not held, so we find the extra guards provided by mandatory locking to be of no significant value.

In Chubby, acquiring a lock in either mode requires write permission so that an unprivileged reader cannot prevent a writer from making progress.

Locking is complex in distributed systems because communication is typically uncertain, and processes may fail independently. Thus, a process holding a lock *L* may issue a request *R*, but then fail. Another process may acquire *L* and perform some action before *R* arrives at its destination. If *R* later arrives, it may be acted on without the protection of *L*, and potentially on inconsistent data. The problem of receiving messages out of order has been well studied; solutions include *virtual time* [11], and *virtual synchrony* [1], which avoids the problem by ensuring that messages are processed in an order consistent with the observations of every participant.

It is costly to introduce sequence numbers into all the interactions in an existing complex system. Instead, Chubby provides a means by which sequence numbers can be introduced into only those interactions that make use of locks. At any time, a lock holder may request a *sequencer*, an opaque byte-string that describes the state of the lock immediately after acquisition. It contains the name of the lock, the mode in which it was acquired (exclusive or shared), and the lock generation number. The client passes the sequencer to servers (such as file servers) if it expects the operation to be protected by the lock. The recipient server is expected to test whether the sequencer is still valid and has the appropriate mode; if not, it should reject the request. The validity of a sequencer can be checked against the server's Chubby cache or, if the server does not wish to maintain a session with Chubby, against the most recent sequencer that the server has observed. The sequencer mechanism requires only the addition of a string to affected messages, and is easily explained to our developers.

Although we find sequencers simple to use, important protocols evolve slowly. Chubby therefore provides an imperfect but easier mechanism to reduce the risk of delayed or re-ordered requests to servers that do not support sequencers. If a client releases a lock in the normal way, it is immediately available for other clients to claim, as one would expect. However, if a lock becomes free because the holder has failed or become inaccessible, the lock server will prevent other clients from claiming the lock for a period called the *lock-delay*. Clients may specify any lock-delay up to some bound, currently one minute; this limit prevents a faulty client from making a lock (and thus some resource) unavailable for an arbitrarily long time. While imperfect, the lock-delay protects unmodified servers and clients from everyday problems caused by message delays and restarts.

2.5 Events

Chubby clients may subscribe to a range of events when they create a handle. These events are delivered to the client asynchronously via an up-call from the Chubby library. Events include:

- file contents modified—often used to monitor the location of a service advertised via the file.
- child node added, removed, or modified—used to implement mirroring (§2.12). (In addition to allowing new files to be discovered, returning events for child nodes makes it possible to monitor ephemeral files without affecting their reference counts.)
- Chubby master failed over—warns clients that other events may have been lost, so data must be rescanned.
- a handle (and its lock) has become invalid—this typically suggests a communications problem.
- lock acquired—can be used to determine when a primary has been elected.
- conflicting lock request from another client—allows the caching of locks.

Events are delivered after the corresponding action has taken place. Thus, if a client is informed that file contents have changed, it is guaranteed to see the new data (or data that is yet more recent) if it subsequently reads the file.

The last two events mentioned are rarely used, and with hindsight could have been omitted. After primary election for example, clients typically need to communicate with the new primary, rather than simply know that a primary exists; thus, they wait for a file modification event indicating that the new primary has written its address in a file. The conflicting lock event in theory permits clients to cache data held on other servers, using Chubby locks to maintain cache consistency. A notification of a conflicting lock request would tell a client to finish using data associated with the lock: it would finish pending operations, flush modifications to a home location, discard cached data, and release. So far, no one has adopted this style of use.

2.6 API

Clients see a Chubby handle as a pointer to an opaque structure that supports various operations. Handles are created only by `Open()`, and destroyed with `Close()`.

`Open()` opens a named file or directory to produce a handle, analogous to a UNIX file descriptor. Only this call takes a node name; all others operate on handles.

The name is evaluated relative to an existing directory handle; the library provides a handle on "/" that is always valid. Directory handles avoid the difficulties of using a program-wide *current directory* in a multi-threaded program that contains many layers of abstraction [18].

The client indicates various options:

- how the handle will be used (reading; writing and locking; changing the ACL); the handle is created only if the client has the appropriate permissions.
- events that should be delivered (see §2.5).
- the lock-delay (§2.4).
- whether a new file or directory should (or must) be created. If a file is created, the caller may supply initial contents and initial ACL names. The return value indicates whether the file was in fact created.

`Close()` closes an open handle. Further use of the handle is not permitted. This call never fails. A related call `Poison()` causes outstanding and subsequent operations on the handle to fail without closing it; this allows a client to cancel Chubby calls made by other threads without fear of deallocating the memory being accessed by them.

The main calls that act on a handle are:

`GetContentsAndStat()` returns both the contents and meta-data of a file. The contents of a file are read atomically and in their entirety. We avoided partial reads and writes to discourage large files. A related call `GetStat()` returns just the meta-data, while `ReadDir()` returns the names and meta-data for the children of a directory.

`SetContents()` writes the contents of a file. Optionally, the client may provide a content generation number to allow the client to simulate compare-and-swap on a file; the contents are changed only if the generation number is current. The contents of a file are always written atomically and in their entirety. A related call `SetACL()` performs a similar operation on the ACL names associated with the node.

`Delete()` deletes the node if it has no children.

`Acquire(), TryAcquire(), Release()` acquire and release locks.

`GetSequencer()` returns a sequencer (§2.4) that describes any lock held by this handle.

`SetSequencer()` associates a sequencer with a handle. Subsequent operations on the handle fail if the sequencer is no longer valid.

`CheckSequencer()` checks whether a sequencer is valid (see §2.4).

Calls fail if the node has been deleted since the handle was created, even if the file has been subsequently recreated. That is, a handle is associated with an instance of a file, rather than with a file name. Chubby may apply access control checks on any call, but always checks `Open()` calls (see §2.3).

All the calls above take an *operation* parameter in addition to any others needed by the call itself. The operation parameter holds data and control information that may be associated with any call. In particular, via the operation parameter the client may:

- supply a callback to make the call asynchronous,
- wait for the completion of such a call, and/or
- obtain extended error and diagnostic information.

Clients can use this API to perform primary election as follows: All potential primaries open the lock file and attempt to acquire the lock. One succeeds and becomes the primary, while the others act as replicas. The primary writes its identity into the lock file with `SetContents()` so that it can be found by clients and replicas, which read the file with `GetContentsAndStat()`, perhaps in response to a file-modification event (§2.5). Ideally, the primary obtains a sequencer with `GetSequencer()`, which it then passes to servers it communicates with; they should confirm with `CheckSequencer()` that it is still the primary. A lock-delay may be used with services that cannot check sequencers (§2.4).

2.7 Caching

To reduce read traffic, Chubby clients cache file data and node meta-data (including file absence) in a consistent, write-through cache held in memory. The cache is maintained by a lease mechanism described below, and kept consistent by invalidations sent by the master, which keeps a list of what each client may be caching. The protocol ensures that clients see either a consistent view of Chubby state, or an error.

When file data or meta-data is to be changed, the modification is blocked while the master sends invalidations for the data to every client that may have cached it; this mechanism sits on top of KeepAlive RPCs, discussed more fully in the next section. On receipt of an invalidation, a client flushes the invalidated state and acknowledges by making its next KeepAlive call. The modification proceeds only after the server knows that each client has invalidated its cache, either because the client acknowledged the invalidation, or because the client allowed its cache lease to expire.

Only one round of invalidations is needed because the master treats the node as *uncachable* while cache invalidations remain unacknowledged. This approach allows reads always to be processed without delay; this is useful because reads greatly outnumber writes. An alternative would be to block calls that access the node during invalidation; this would make it less likely that over-eager clients will bombard the master with uncached accesses during invalidation, at the cost of occasional delays. If this were a problem, one could imagine adopting a hybrid scheme that switched tactics if overload were detected.

The caching protocol is simple: it invalidates cached data on a change, and never updates it. It would be just as simple to update rather than to invalidate, but update-only protocols can be arbitrarily inefficient; a client that accessed a file might receive updates indefinitely, causing an unbounded number of unnecessary updates.

Despite the overheads of providing strict consistency, we rejected weaker models because we felt that programmers would find them harder to use. Similarly, mechanisms such as virtual synchrony that require clients to exchange sequence numbers in all messages were considered inappropriate in an environment with diverse pre-existing communication protocols.

In addition to caching data and meta-data, Chubby clients cache open handles. Thus, if a client opens a file it has opened previously, only the first `Open()` call necessarily leads to an RPC to the master. This caching is restricted in minor ways so that it never affects the semantics observed by the client: handles on ephemeral files cannot be held open if the application has closed them; and handles that permit locking can be reused, but cannot be used concurrently by multiple application handles. This last restriction exists because the client may use `Close()` or `Poison()` for their side-effect of cancelling outstanding `Acquire()` calls to the master.

Chubby's protocol permits clients to cache locks—that is, to hold locks longer than strictly necessary in the hope that they can be used again by the same client. An event informs a lock holder if another client has requested a conflicting lock, allowing the holder to release the lock just when it is needed elsewhere (see §2.5).

2.8 Sessions and KeepAlives

A Chubby session is a relationship between a Chubby cell and a Chubby client; it exists for some interval of time, and is maintained by periodic handshakes called KeepAlives. Unless a Chubby client informs the master otherwise, the client's handles, locks, and cached data all remain valid provided its session remains valid. (However, the protocol for session maintenance may require the client to acknowledge a cache invalidation in order to maintain its session; see below.)

A client requests a new session on first contacting the master of a Chubby cell. It ends the session explicitly either when it terminates, or if the session has been idle (with no open handles and no calls for a minute).

Each session has an associated lease—an interval of time extending into the future during which the master guarantees not to terminate the session unilaterally. The end of this interval is called the session lease timeout. The master is free to advance this timeout further into the future, but may not move it backwards in time.

The master advances the lease timeout in three circumstances: on creation of the session, when a master fail-over occurs (see below), and when it responds to a KeepAlive RPC from the client. On receiving a KeepAlive, the master typically blocks the RPC (does not allow it to return) until the client's previous lease interval is close to expiring. The master later allows the RPC to return to the client, and thus informs the client of the new lease timeout. The master may extend the timeout by any amount. The default extension is 12s, but an overloaded master may use higher values to reduce the number of KeepAlive calls it must process. The client initiates a new KeepAlive immediately after receiving the previous reply. Thus, the client ensures that there is almost always a KeepAlive call blocked at the master.

As well as extending the client's lease, the KeepAlive reply is used to transmit events and cache invalidations back to the client. The master allows a KeepAlive to return early when an event or invalidation is to be delivered. Piggybacking events on KeepAlive replies ensures that clients cannot maintain a session without acknowledging cache invalidations, and causes all Chubby RPCs to flow from client to master. This simplifies the client, and allows the protocol to operate through firewalls that allow initiation of connections in only one direction.

The client maintains a local lease timeout that is a conservative approximation of the master's lease timeout. It differs from the master's lease timeout because the client must make conservative assumptions both of the time its KeepAlive reply spent in flight, and the rate at which the master's clock is advancing; to maintain consistency, we require that the server's clock advance no faster than a known constant factor faster than the client's.

If a client's local lease timeout expires, it becomes unsure whether the master has terminated its session. The client empties and disables its cache, and we say that its session is in *jeopardy*. The client waits a further interval called the grace period, 45s by default. If the client and master manage to exchange a successful KeepAlive be-

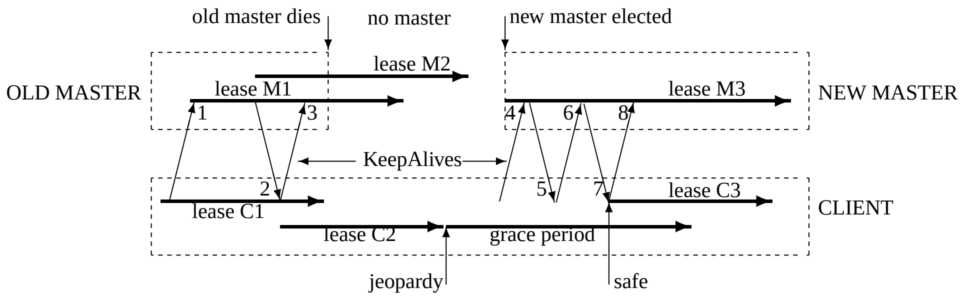


Figure 2: The role of the grace period in master fail-over

fore the end of the client's grace period, the client enables its cache once more. Otherwise, the client assumes that the session has expired. This is done so that Chubby API calls do not block indefinitely when a Chubby cell becomes inaccessible; calls return with an error if the grace period ends before communication is re-established.

The Chubby library can inform the application when the grace period begins via a *jeopardy* event. When the session is known to have survived the communications problem, a *safe* event tells the client to proceed; if the session times out instead, an *expired* event is sent. This information allows the application to quiesce itself when it is unsure of the status of its session, and to recover without restarting if the problem proves to be transient. This can be important in avoiding outages in services with large startup overhead.

If a client holds a handle *H* on a node and any operation on *H* fails because the associated session has expired, all subsequent operations on *H* (except `Close()` and `Poison()`) will fail in the same way. Clients can use this to guarantee that network and server outages cause only a suffix of a sequence of operations to be lost, rather than an arbitrary subsequence, thus allowing complex changes to be marked as committed with a final write.

2.9 Fail-overs

When a master fails or otherwise loses mastership, it discards its in-memory state about sessions, handles, and locks. The authoritative timer for session leases runs at the master, so until a new master is elected the session lease timer is stopped; this is legal because it is equivalent to extending the client's lease. If a master election occurs quickly, clients can contact the new master before their local (approximate) lease timers expire. If the election takes a long time, clients flush their caches and wait for the grace period while trying to find the new master. Thus the grace period allows sessions to be maintained across fail-overs that exceed the normal lease timeout.

Figure 2 shows the sequence of events in a lengthy master fail-over event in which the client must use its grace period to preserve its session. Time increases from left to right, but times are not to scale. Client session leases are shown as thick arrows both as viewed by both the old and new masters (M1-3, above) and the client (C1-3, below). Upward angled arrows indicate KeepAlive requests, and downward angled arrows their replies. The original master has session lease M1 for the client, while the client has a conservative approximation C1. The master commits to lease M2 before informing the client via KeepAlive reply 2; the client is able to extend its view of the lease C2. The master dies before replying to the next KeepAlive, and some time elapses before another master is elected. Eventually the client's approximation of its lease (C2) expires. The client then flushes its cache and starts a timer for the grace period.

During this period, the client cannot be sure whether its lease has expired at the master. It does not tear down its session, but it blocks all application calls on its API to prevent the application from observing inconsistent data. At the start of the grace period, the Chubby library sends a *jeopardy* event to the application to allow it to quiesce itself until it can be sure of the status of its session.

Eventually a new master election succeeds. The master initially uses a conservative approximation M3 of the session lease that its predecessor may have had for the client. The first KeepAlive request (4) from the client to the new master is rejected because it has the wrong master epoch number (described in detail below). The retried request (6) succeeds but typically does not extend the master lease further because M3 was conservative. However the reply (7) allows the client to extend its lease (C3) once more, and optionally inform the application that its session is no longer in jeopardy. Because the grace period was long enough to cover the interval between the end of lease C2 and the beginning of lease C3, the client saw nothing but a delay. Had the grace period been less than that interval, the client would have abandoned the session and reported the failure to the application.

Once a client has contacted the new master, the client library and master co-operate to provide the illusion to the application that no failure has occurred. To achieve this, the new master must reconstruct a conservative approximation of the in-memory state that the previous master had. It does this partly by reading data stored stably on disc (replicated via the normal database replication protocol), partly by obtaining state from clients, and partly by conservative assumptions. The database records each session, held lock, and ephemeral file.

A newly elected master proceeds:

1. It first picks a new client *epoch number*, which clients are required to present on every call. The master rejects calls from clients using older epoch numbers, and provides the new epoch number. This ensures that the new master will not respond to a very old packet that was sent to a previous master, even one running on the same machine.
2. The new master may respond to master-location requests, but does not at first process incoming session-related operations.
3. It builds in-memory data structures for sessions and locks that are recorded in the database. Session leases are extended to the maximum that the previous master may have been using.
4. The master now lets clients perform KeepAlives, but no other session-related operations.
5. It emits a fail-over event to each session; this causes clients to flush their caches (because they may have missed invalidations), and to warn applications that other events may have been lost.
6. The master waits until each session acknowledges the fail-over event or lets its session expire.
7. The master allows all operations to proceed.
8. If a client uses a handle created prior to the fail-over (determined from the value of a sequence number in the handle), the master recreates the in-memory representation of the handle and honours the call. If such a recreated handle is closed, the master records it in memory so that it cannot be recreated in this master epoch; this ensures that a delayed or duplicated network packet cannot accidentally recreate a closed handle. A faulty client can recreate a closed handle in a future epoch, but this is harmless given that the client is already faulty.
9. After some interval (a minute, say), the master deletes ephemeral files that have no open file handles. Clients should refresh handles on ephemeral files during this interval after a fail-over. This mechanism has the unfortunate effect that ephemeral files may not disappear promptly if the last client on such a file loses its session during a fail-over.

Readers will be unsurprised to learn that the fail-over code, which is exercised far less often than other parts of the system, has been a rich source of interesting bugs.

2.10 Database implementation

The first version of Chubby used the replicated version of Berkeley DB [20] as its database. Berkeley DB provides B-trees that map byte-string keys to arbitrary byte-string values. We installed a key comparison function that sorts first by the number of components in a path name; this allows nodes to by keyed by their path name, while keeping sibling nodes adjacent in the sort order. Because Chubby does not use path-based permissions, a single lookup in the database suffices for each file access.

Berkeley DB's uses a distributed consensus protocol to replicate its database logs over a set of servers. Once master leases were added, this matched the design of Chubby, which made implementation straightforward.

While Berkeley DB's B-tree code is widely-used and mature, the replication code was added recently, and has fewer users. Software maintainers must give priority to maintaining and improving their most popular product features. While Berkeley DB's maintainers solved the problems we had, we felt that use of the replication code exposed us to more risk than we wished to take. As a result, we have written a simple database using write ahead logging and snapshotting similar to the design of Birrell *et al.* [2]. As before, the database log is distributed among the replicas using a distributed consensus protocol. Chubby used few of the features of Berkeley DB, and so this rewrite allowed significant simplification of the system as a whole; for example, while we needed atomic operations, we did not need general transactions.

2.11 Backup

Every few hours, the master of each Chubby cell writes a snapshot of its database to a GFS file server [7] in a different building. The use of a separate building ensures both that the backup will survive building damage, and that the backups introduce no cyclic dependencies in the system; a GFS cell in the same building potentially might rely on the Chubby cell for electing its master.

Backups provide both disaster recovery and a means for initializing the database of a newly replaced replica without placing load on replicas that are in service.

2.12 Mirroring

Chubby allows a collection of files to be mirrored from one cell to another. Mirroring is fast because the files are small and the event mechanism (§2.5) informs the mirroring code immediately if a file is added, deleted, or modified. Provided there are no network problems, changes are reflected in dozens of mirrors world-wide in well under a second. If a mirror is unreachable, it remains unchanged until connectivity is restored. Updated files are then identified by comparing their checksums.

Mirroring is used most commonly to copy configuration files to various computing clusters distributed around the world. A special cell, named `global`, contains a subtree `/ls/global/master` that is mirrored to the subtree `/ls/`*cell*`/slave` in every other Chubby cell. The global cell is special because its five replicas are located in widely-separated parts of the world, so it is almost always accessible from most of the organization.

Among the files mirrored from the global cell are Chubby's own access control lists, various files in which Chubby cells and other systems advertise their presence to our monitoring services, pointers to allow clients to locate large data sets such as Bigtable cells, and many configuration files for other systems.

3 Mechanisms for scaling

Chubby's clients are individual processes, so Chubby must handle more clients than one might expect; we have seen 90,000 clients communicating directly with a Chubby master—far more than the number of machines involved. Because there is just one master per cell, and its machine is identical to those of the clients, the clients can overwhelm the master by a huge margin. Thus, the most effective scaling techniques reduce communication with the master by a significant factor. Assuming the master has no serious performance bug, minor improvements in request processing at the master have little effect. We use several approaches:

- We can create an arbitrary number of Chubby cells; clients almost always use a nearby cell (found with DNS) to avoid reliance on remote machines. Our typical deployment uses one Chubby cell for a data centre of several thousand machines.
- The master may increase lease times from the default 12s up to around 60s when it is under heavy load, so it need process fewer KeepAlive RPCs. (KeepAlives are *by far* the dominant type of request (see 4.1), and failure to process them in time is the typical failure mode of an overloaded server; clients are largely insensitive to latency variation in other calls.)
- Chubby clients cache file data, meta-data, the absence of files, and open handles to reduce the number of calls they make on the server.
- We use protocol-conversion servers that translate the Chubby protocol into less-complex protocols such as DNS and others. We discuss some of these below.

Here we describe two familiar mechanisms, proxies and partitioning, that we expect will allow Chubby to scale further. We do not yet use them in production, but they are designed, and may be used soon. We have no present need to consider scaling beyond a factor of five: First, there are limits on the number of machines one would wish to put in a data centre or make reliant on a single instance of a service. Second, because we use similar machines for Chubby clients and servers, hardware improvements that increase the number of clients per machine also increase the capacity of each server.

3.1 Proxies

Chubby's protocol can be proxied (using the same protocol on both sides) by trusted processes that pass requests from other clients to a Chubby cell. A proxy can reduce server load by handling both KeepAlive and read requests; it cannot reduce write traffic, which passes through the proxy's cache. But even with aggressive client caching, write traffic constitutes much less than one percent of Chubby's normal workload (see §4.1), so proxies allow a significant increase in the number of clients. If a proxy handles N_{proxy} clients, KeepAlive traffic is reduced by a factor of N_{proxy}, which might be 10 thousand or more. A proxy cache can reduce read traffic by at most the mean amount of read-sharing—a factor of around 10 (§4.1). But because reads constitute under 10% of Chubby's load at present, the saving in KeepAlive traffic is by far the more important effect.

Proxies add an additional RPC to writes and first-time reads. One might expect proxies to make the cell temporarily unavailable at least twice as often as before, because each proxied client depends on two machines that may fail: its proxy and the Chubby master.

Alert readers will notice that the fail-over strategy described in Section 2.9, is not ideal for proxies. We discuss this problem in Section 4.4.

3.2 Partitioning

As mentioned in Section 2.3, Chubby's interface was chosen so that the name space of a cell could be partitioned between servers. Although we have not yet needed it, the code can partition the name space by directory. If enabled, a Chubby cell would be composed of N *partitions*, each of which has a set of replicas and a master. Every node D/C in directory D would be stored on the partition $P(D/C) = \mathtt{hash}(D) \bmod N$. Note that the meta-data for D may be stored on a different partition $P(D) = \mathtt{hash}(D') \bmod N$, where D' is the parent of D.

Partitioning is intended to enable large Chubby cells with little communication between the partitions. Although Chubby lacks hard links, directory modified-times, and cross-directory rename operations, a few operations still require cross-partition communication:

- ACLs are themselves files, so one partition may use another for permissions checks. However, ACL files are readily cached; only `Open()` and `Delete()` calls require ACL checks; and most clients read publicly accessible files that require no ACL.
- When a directory is deleted, a cross-partition call may be needed to ensure that the directory is empty.

Because each partition handles most calls independently of the others, we expect this communication to have only a modest impact on performance or availability.

Unless the number of partitions N is large, one would expect that each client would contact the majority of the partitions. Thus, partitioning reduces read and write traffic on any given partition by a factor of N but does not necessarily reduce KeepAlive traffic. Should it be necessary for Chubby to handle more clients, our strategy involves a combination of proxies and partitioning.

4 Use, surprises and design errors

4.1 Use and behaviour

The following table gives statistics taken as a snapshot of a Chubby cell; the RPC rate was a seen over a ten-minute period. The numbers are typical of cells in Google.

time since last fail-over	18 days
fail-over duration	14s
active clients (direct)	22k
additional proxied clients	32k
files open	12k
naming-related	60%
client-is-caching-file entries	230k
distinct files cached	24k
names negatively cached	32k
exclusive locks	1k
shared locks	0
stored directories	8k
ephemeral	0.1%
stored files	22k
0-1k bytes	90%
1k-10k bytes	10%
$>$ 10k bytes	0.2%
naming-related	46%
mirrored ACLs & config info	27%
GFS and Bigtable meta-data	11%
ephemeral	3%
RPC rate	1-2k/s
KeepAlive	93%
GetStat	2%
Open	1%
CreateSession	1%
GetContentsAndStat	0.4%
SetContents	680ppm
Acquire	31ppm

Several things can be seen:

- Many files are used for naming; see §4.3.
- Configuration, access control, and meta-data files (analogous to file system super-blocks) are common.
- Negative caching is significant.
- 230k/24k≈10 clients use each cached file, on average.
- Few clients hold locks, and shared locks are rare; this is consistent with locking being used for primary election and partitioning data among replicas.
- RPC traffic is dominated by session KeepAlives; there are a few reads (which are cache misses); there are very few writes or lock acquisitions.

Now we briefly describe the typical causes of outages in our cells. If we assume (optimistically) that a cell is "up" if it has a master that is willing to serve, on a sample of our cells we recorded 61 outages over a period of a few weeks, amounting to 700 cell-days of data in total. We excluded outages due to maintenance that shut down the data centre. All other causes are included: network congestion, maintenance, overload, and errors due to operators, software, and hardware. Most outages were 15s or less, and 52 were under 30s; most of our applications are not affected significantly by Chubby outages under 30s. The remaining nine outages were caused by network maintenance (4), suspected network connectivity problems (2), software errors (2), and overload (1).

In a few dozen cell-years of operation, we have lost data on six occasions, due to database software errors (4) and operator error (2); none involved hardware error. Ironically, the operational errors involved upgrades to avoid the software errors. We have twice corrected corruptions caused by software in non-master replicas.

Chubby's data fits in RAM, so most operations are cheap. Mean request latency at our production servers is consistently a small fraction of a millisecond regardless of cell load until the cell approaches overload, when latency increases dramatically and sessions are dropped. Overload typically occurs when many sessions ($> 90,000$) are active, but can result from exceptional conditions: when clients made millions of read requests simultaneously (described in Section 4.3), and when a mistake in the client library disabled caching for some reads, resulting in tens of thousands of requests per second. Because most RPCs are KeepAlives, the server can maintain a low mean request latency with many active clients by increasing the session lease period (see §3). Group commit reduces the effective work done per request when bursts of writes arrive, but this is rare.

RPC read latencies measured at the client are limited by the RPC system and network; they are under 1ms for a local cell, but 250ms between antipodes. Writes (which include lock operations) are delayed a further 5-10ms by the database log update, but by up to tens of seconds if a recently-failed client cached the file. Even this variability in write latency has little effect on the mean request latency at the server because writes are so infrequent.

Clients are fairly insensitive to latency variation provided sessions are not dropped. At one point, we added artificial delays in `Open()` to curb abusive clients (see §4.5); developers noticed only when delays exceeded ten seconds *and* were applied repeatedly. We have found that the key to scaling Chubby is not server performance; reducing communication to the server can have far greater impact. No significant effort has been applied to tuning read/write server code paths; we checked that no egregious bugs were present, then focused on the scaling mechanisms that could be more effective. On the other hand, developers do notice if a performance bug affects the local Chubby cache, which a client may read thousands of times per second.

4.2 Java clients

Google's infrastructure is mostly in C++, but a growing number of systems are being written in Java [8]. This trend presented an unanticipated problem for Chubby, which has a complex client protocol and a non-trivial client-side library.

Java encourages portability of entire applications at the expense of incremental adoption by making it somewhat irksome to link against other languages. The usual Java mechanism for accessing non-native libraries is JNI [15], but it is regarded as slow and cumbersome. Our Java programmers so dislike JNI that to avoid its use they prefer to translate large libraries into Java, and commit to supporting them.

Chubby's C++ client library is 7000 lines (comparable with the server), and the client protocol is delicate. To maintain the library in Java would require care and expense, while an implementation without caching would burden the Chubby servers. Thus our Java users run copies of a protocol-conversion server that exports a simple RPC protocol that corresponds closely to Chubby's client API. Even with hindsight, it is not obvious how we might have avoided the cost of writing, running and maintaining this additional server.

4.3 Use as a name service

Even though Chubby was designed as a lock service, we found that its most popular use was as a name server.

Caching within the normal Internet naming system, the DNS, is based on time. DNS entries have a *time-to-live* (TTL), and DNS data are discarded when they have not been refreshed within that period. Usually it is straightforward to pick a suitable TTL value, but if prompt replacement of failed services is desired, the TTL can become small enough to overload the DNS servers.

For example, it is common for our developers to run jobs involving thousands of processes, and for each process to communicate with every other, leading to a quadratic number of DNS lookups. We might wish to use a TTL of 60s; this would allow misbehaving clients to be replaced without excessive delay and is not considered an unreasonably short replacement time in our environment. In that case, to maintain the DNS caches of a single job as small as 3 thousand clients would require 150 thousand lookups per second. (For comparison, a 2-CPU 2.6GHz Xeon DNS server might handle 50 thousand requests per second.) Larger jobs create worse problems, and several jobs many be running at once. The variability in our DNS load had been a serious problem for Google before Chubby was introduced.

In contrast, Chubby's caching uses explicit invalidations so a constant rate of session KeepAlive requests can maintain an arbitrary number of cache entries indefinitely at a client, in the absence of changes. A 2-CPU 2.6GHz Xeon Chubby master has been seen to handle 90 thousand clients communicating directly with it (no proxies); the clients included large jobs with communication patterns of the kind described above. The ability to provide swift name updates without polling each name individually is so appealing that Chubby now provides name service for most of the company's systems.

Although Chubby's caching allows a single cell to sustain a large number of clients, load spikes can still be a problem. When we first deployed the Chubby-based name service, starting a 3 thousand process job (thus generating 9 million requests) could bring the Chubby master to its knees. To resolve this problem, we chose to group name entries into batches so that a single lookup would return and cache the name mappings for a large number (typically 100) of related processes within a job.

The caching semantics provided by Chubby are more precise than those needed by a name service; name resolution requires only timely notification rather than full consistency. As a result, there was an opportunity for reducing the load on Chubby by introducing a simple protocol-conversion server designed specifically for name lookups. Had we foreseen the use of Chubby as a name service, we might have chosen to implement full proxies sooner than we did in order to avoid the need for this simple, but nevertheless additional server.

One further protocol-conversion server exists: the Chubby DNS server. This makes the naming data stored within Chubby available to DNS clients. This server is important both for easing the transition from DNS names to Chubby names, and to accommodate existing applications that cannot be converted easily, such as browsers.

4.4 Problems with fail-over

The original design for master fail-over (§2.9) requires the master to write new sessions to the database as they are created. In the Berkeley DB version of the lock server, the overhead of creating sessions became a problem when many processes were started at once. To avoid overload, the server was modified to store a session in the database not when it was first created, but instead when it attempted its first modification, lock acquisition, or open of an ephemeral file. In addition, active sessions were recorded in the database with some probability on each KeepAlive. Thus, the writes for read-only sessions were spread out in time.

Though it was necessary to avoid overload, this optimization has the undesirable effect that young read-only sessions may not be recorded in the database, and so may be discarded if a fail-over occurs. Although such sessions hold no locks, this is unsafe; if all the recorded sessions were to check in with the new master before the leases of discarded sessions expired, the discarded sessions could then read stale data for a while. This is rare in practice; in a large system it is almost certain that some session will fail to check in, and thus force the new master to await the maximum lease time anyway. Nevertheless, we have modified the fail-over design both to avoid this effect, and to avoid a complication that the current scheme introduces to proxies.

Under the new design, we avoid recording sessions in the database at all, and instead recreate them in the same way that the master currently recreates handles (§2.9,¶8). A new master must now wait a full worst-case lease timeout before allowing operations to proceed, since it cannot know whether all sessions have checked in (§2.9,¶6). Again, this has little effect in practice because it is likely that not all sessions will check in.

Once sessions can be recreated without on-disc state, proxy servers can manage sessions that the master is not aware of. An extra operation available only to proxies allows them to change the session that locks are associated with. This permits one proxy to take over a client from another when a proxy fails. The only further change needed at the master is a guarantee not to relinquish locks or ephemeral file handles associated with proxy sessions until a new proxy has had a chance to claim them.

4.5 Abusive clients

Google's project teams are free to set up their own Chubby cells, but doing so adds to their maintenance burden, and consumes additional hardware resources. Many services therefore use shared Chubby cells, which makes it important to isolate clients from the misbehaviour of others. Chubby is intended to operate within a single company, and so malicious denial-of-service attacks against it are rare. However, mistakes, misunderstandings, and the differing expectations of our developers lead to effects that are similar to attacks.

Some of our remedies are heavy-handed. For example, we review the ways project teams plan to use Chubby, and deny access to the shared Chubby name space until review is satisfactory. A problem with this approach is that developers are often unable to predict how their services will be used in the future, and how use will grow. Readers will note the irony of our own failure to predict how Chubby itself would be used.

The most important aspect of our review is to determine whether use of any of Chubby's resources (RPC rate, disc space, number of files) grows linearly (or worse) with number of users or amount of data processed by the project. Any linear growth must be mitigated by a compensating parameter that can be adjusted to reduce the load on Chubby to reasonable bounds. Nevertheless our early reviews were not thorough enough.

A related problem is the lack of performance advice in most software documentation. A module written by one team may be reused a year later by another team with disastrous results. It is sometimes hard to explain to interface designers that they must change their interfaces not because they are bad, but because other developers may be less aware of the cost of an RPC.

Below we list some problem cases we encountered.

Lack of aggressive caching Originally, we did not appreciate the critical need to cache the absence of files, nor to reuse open file handles. Despite attempts at education, our developers regularly write loops that retry indefinitely when a file is not present, or poll a file by opening it and closing it repeatedly when one might expect they would open the file just once.

At first we countered these retry-loops by introducing exponentially-increasing delays when an application made many attempts to `Open()` the same file over a short period. In some cases this exposed bugs that developers acknowledged, but often it required us to spend yet more time on education. In the end it was easier to make repeated `Open()` calls cheap.

Lack of quotas Chubby was never intended to be used as a storage system for large amounts of data, and so it has no storage quotas. In hindsight, this was naïve.

One of Google's projects wrote a module to keep track of data uploads, storing some meta-data in Chubby. Such uploads occurred rarely and were limited to a small set of people, so the space was bounded. However, two other services started using the same module as a means for tracking uploads from a wider population of users. Inevitably, these services grew until the use of Chubby was extreme: a single 1.5MByte file was being rewritten in its entirety on each user action, and the overall space used by the service exceeded the space needs of all other Chubby clients combined.

We introduced a limit on file size (256kBytes), and encouraged the services to migrate to more appropriate storage systems. But it is difficult to make significant changes to production systems maintained by busy people—it took approximately a year for the data to be migrated elsewhere.

Publish/subscribe There have been several attempts to use Chubby's event mechanism as a publish/subscribe system in the style of Zephyr [6]. Chubby's heavyweight guarantees and its use of invalidation rather than update in maintaining cache consistency make it a slow and inefficient for all but the most trivial publish/subscribe examples. Fortunately, all such uses have been caught before the cost of redesigning the application was too large.

4.6 Lessons learned

Here we list lessons, and miscellaneous design changes we might make if we have the opportunity:

Developers rarely consider availability We find that our developers rarely think about failure probabilities, and are inclined to treat a service like Chubby as though it were always available. For example, our developers once built a system employing hundred of machines that initiated recovery procedures taking tens of minutes when Chubby elected a new master. This magnified the consequences of a single failure by a factor of a hundred both in time *and* the number of machines affected. We would prefer developers to plan for short Chubby outages, so that such an event has little or no affect on their applications. This is one of the arguments for coarse-grained locking, discussed in Section 2.1.

Developers also fail to appreciate the difference between a service being up, and that service being available to their applications. For example, the global Chubby cell (see §2.12), is almost always up because it is rare for more than two geographically distant data centres to be down simultaneously. However, its *observed availability for a given client* is usually lower than the observed availability of the client's local Chubby cell. First, the local cell is less likely to be partitioned from the client, and second, although the local cell may be down often due to maintenance, the same maintenance affects the client directly, so Chubby's unavailability is not observed by the client.

Our API choices can also affect the way developers chose to handle Chubby outages. For example, Chubby provides an event that allows clients to detect when a master fail-over has taken place. The intent was for clients to check for possible changes, as other events may have been lost. Unfortunately, many developers chose to crash their applications on receiving this event, thus decreasing the availability of their systems substantially. We might have done better to send redundant "file change" events instead, or even to ensure that no events were lost during a fail-over.

At present we use three mechanisms to prevent developers from being over-optimistic about Chubby availability, especially that of the global cell. First, as previously mentioned (§4.5), we review how project teams plan to use Chubby, and advise them against techniques that would tie their availability too closely to Chubby's. Second, we now supply libraries that perform some high-level tasks so that developers are automatically isolated from Chubby outages. Third, we use the post-mortem of each Chubby outage as a means not only of eliminating bugs in Chubby and our operational procedures, but of reducing the sensitivity of applications to Chubby's availability—both can lead to better availability of our systems overall.

Fine-grained locking could be ignored At the end of Section 2.1 we sketched a design for a server that clients could run to provide fine-grained locking. It is perhaps a surprise that so far we have not needed to write such a server; our developers typically find that to optimize their applications, they must remove unnecessary communication, and that often means finding a way to use coarse-grained locking.

Poor API choices have unexpected affects For the most part, our API has evolved well, but one mistake stands out. Our means for cancelling long-running calls are the `Close()` and `Poison()` RPCs, which also discard the server state for the handle. This prevents handles that can acquire locks from being shared, for example, by multiple threads. We may add a `Cancel()` RPC to allow more sharing of open handles.

RPC use affects transport protocols KeepAlives are used both for refreshing the client's session lease, and for passing events and cache invalidations from the master to the client. This design has the automatic and desirable consequence that a client cannot refresh its session lease without acknowledging cache invalidations.

This would seem ideal, except that it introduced a tension in our choice of protocol. TCP's back off policies pay no attention to higher-level timeouts such as Chubby leases, so TCP-based KeepAlives led to many lost sessions at times of high network congestion. We were forced to send KeepAlive RPCs via UDP rather than TCP; UDP has no congestion avoidance mechanisms, so we would prefer to use UDP only when high-level time-bounds must be met.

We may augment the protocol with an additional TCP-based `GetEvent()` RPC which would be used to communicate events and invalidations in the normal case, used in the same way KeepAlives. The KeepAlive reply would still contain a list of unacknowledged events so that events must eventually be acknowledged.

5 Comparison with related work

Chubby is based on well-established ideas. Chubby's cache design is derived from work on distributed file systems [10]. Its sessions and cache tokens are similar in behaviour to those in Echo [17]; sessions reduce the overhead of leases [9] in the V system. The idea of exposing a general-purpose lock service is found in VMS [23], though that system initially used a special-purpose high-speed interconnect that permitted low-latency interactions. Like its caching model, Chubby's API is based on a file-system model, including the idea that a file-system-like name space is convenient for more than just files [18, 21, 22].

Chubby differs from a distributed file system such as Echo or AFS [10] in its performance and storage aspirations: Clients do not read, write, or store large amounts of data, and they do not expect high throughput or even low-latency unless the data is cached. They do expect consistency, availability, and reliability, but these attributes are easier to achieve when performance is less important. Because Chubby's database is small, we are able to store many copies of it on-line (typically five replicas and a few backups). We take full backups multiple times per day, and via checksums of the database state, we compare replicas with one another every few hours. The weakening of the normal file system performance and storage requirements allows us to serve tens of thousands of clients from a single Chubby master. By providing a central point where many clients can share information and co-ordinate activities, we have solved a class of problems faced by our system developers.

The large number of file systems and lock servers described in the literature prevents an exhaustive comparison, so we provide details on one: we chose to compare with Boxwood's lock server [16] because it was designed recently, it too is designed to run in a loosely-coupled environment, and yet its design differs in various ways from Chubby, some interesting and some incidental.

Chubby implements locks, a reliable small-file storage system, and a session/lease mechanism in a single service. In contrast, Boxwood separates these into three: a *lock service*, a *Paxos service* (a reliable repository for state), and a *failure detection service* respectively. The Boxwood system itself uses these three components together, but another system could use these building blocks independently. We suspect that this difference in design stems from a difference in target audience. Chubby was intended for a diverse audience and application mix; its users range from experts who create new distributed systems, to novices who write administration scripts. For our environment, a large-scale shared service with a familiar API seemed attractive. In contrast, Boxwood provides a toolkit that (to our eyes, at least) is appropriate for a smaller number of more sophisticated developers working on projects that may share code but need not be used together.

In many cases, Chubby provides a higher-level interface than Boxwood. For example, Chubby combines the lock and file names spaces, while Boxwood's lock names are simple byte sequences. Chubby clients cache file state by default; a client of Boxwood's Paxos service could implement caching via the lock service, but would probably use the caching provided by Boxwood itself.

The two systems have markedly different default parameters, chosen for different expectations: Each Boxwood failure detector is contacted by each client every 200ms with a timeout of 1s; Chubby's default lease time is 12s and KeepAlives are exchanged every 7s. Boxwood's subcomponents use two or three replicas to achieve availability, while we typically use five replicas per cell. However, these choices alone do not suggest a deep design difference, but rather an indication of how parameters in such systems must be adjusted to accommodate more client machines, or the uncertainties of racks shared with other projects.

A more interesting difference is the introduction of Chubby's grace period, which Boxwood lacks. (Recall that the grace period allows clients to ride out long Chubby master outages without losing sessions or locks. Boxwood's "grace period" is the equivalent of Chubby's "session lease", a different concept.) Again, this difference is the result of differing expectations about scale and failure probability in the two systems. Although master fail-overs are rare, a lost Chubby lock is expensive for clients.

Finally, locks in the two systems are intended for different purposes. Chubby locks are heavier-weight, and need sequencers to allow externals resources to be protected safely, while Boxwood locks are lighter-weight, and intended primarily for use within Boxwood.

6 Summary

Chubby is a distributed lock service intended for coarse-grained synchronization of activities within Google's distributed systems; it has found wider use as a name service and repository for configuration information.

Its design is based on well-known ideas that have meshed well: distributed consensus among a few replicas for fault tolerance, consistent client-side caching to reduce server load while retaining simple semantics, timely notification of updates, and a familiar file system interface. We use caching, protocol-conversion servers, and simple load adaptation to allow it scale to tens of thousands of client processes per Chubby instance. We expect to scale it further via proxies and partitioning.

Chubby has become Google's primary internal name service; it is a common rendezvous mechanism for systems such as MapReduce [4]; the storage systems GFS and Bigtable use Chubby to elect a primary from redundant replicas; and it is a standard repository for files that require high availability, such as access control lists.

7 Acknowledgments

Many contributed to the Chubby system: Sharon Perl wrote the replication layer on Berkeley DB; Tushar Chandra and Robert Griesemer wrote the replicated database that replaced Berkeley DB; Ramsey Haddad connected the API to Google's file system interface; Dave Presotto, Sean Owen, Doug Zongker and Praveen Tamara wrote the Chubby DNS, Java, and naming protocol-converters, and the full Chubby proxy respectively; Vadim Furman added the caching of open handles and file-absence; Rob Pike, Sean Quinlan and Sanjay Ghemawat gave valuable design advice; and many Google developers uncovered early weaknesses.

References

[1] BIRMAN, K. P., AND JOSEPH, T. A. Exploiting virtual synchrony in distributed systems. In *11th SOSP* (1987), pp. 123–138.

[2] BIRRELL, A., JONES, M. B., AND WOBBER, E. A simple and efficient implementation for small databases. In *11th SOSP* (1987), pp. 149–154.

[3] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. Bigtable: A distributed structured data storage system. In *7th OSDI* (2006).

[4] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *6th OSDI* (2004), pp. 137–150.

[5] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (April 1985), 374–382.

[6] FRENCH, R. S., AND KOHL, J. T. *The Zephyr Programmer's Manual*. MIT Project Athena, Apr. 1989.

[7] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *19th SOSP* (Dec. 2003), pp. 29–43.

[8] GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. *Java Language Spec. (2nd Ed.)*. Addison-Wesley, 2000.

[9] GRAY, C. G., AND CHERITON, D. R. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *12th SOSP* (1989), pp. 202–210.

[10] HOWARD, J., KAZAR, M., MENEES, S., NICHOLS, D., SATYANARAYANAN, M., SIDEBOTHAM, R., AND WEST, M. Scale and performance in a distributed file system. *ACM TOCS 6*, 1 (Feb. 1988), 51–81.

[11] JEFFERSON, D. Virtual time. *ACM TOPLAS*, 3 (1985), 404–425.

[12] LAMPORT, L. The part-time parliament. *ACM TOCS 16*, 2 (1998), 133–169.

[13] LAMPORT, L. Paxos made simple. *ACM SIGACT News 32*, 4 (2001), 18–25.

[14] LAMPSON, B. W. How to build a highly available system using consensus. In *Distributed Algorithms*, vol. 1151 of *LNCS*. Springer–Verlag, 1996, pp. 1–17.

[15] LIANG, S. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley, 1999.

[16] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *6th OSDI* (2004), pp. 105–120.

[17] MANN, T., BIRRELL, A., HISGEN, A., JERIAN, C., AND SWART, G. A coherent distributed file cache with directory write-behind. *TOCS 12*, 2 (1994), 123–164.

[18] MCJONES, P., AND SWART, G. Evolving the UNIX system interface to support multithreaded programs. Tech. Rep. 21, DEC SRC, 1987.

[19] OKI, B., AND LISKOV, B. Viewstamped replication: A general primary copy method to support highly-available distributed systems. In *ACM PODC* (1988).

[20] OLSON, M. A., BOSTIC, K., AND SELTZER, M. Berkeley DB. In *USENIX* (June 1999), pp. 183–192.

[21] PIKE, R., PRESOTTO, D. L., DORWARD, S., FLANDRENA, B., THOMPSON, K., TRICKEY, H., AND WINTERBOTTOM, P. Plan 9 from Bell Labs. *Computing Systems 8*, 2 (1995), 221–254.

[22] RITCHIE, D. M., AND THOMPSON, K. The UNIX timesharing system. *CACM 17*, 7 (1974), 365–375.

[23] SNAMAN, JR., W. E., AND THIEL, D. W. The VAX/VMS distributed lock manager. *Digital Technical Journal 1*, 5 (Sept. 1987), 29–44.

[24] YIN, J., MARTIN, J.-P., VENKATARAMANI, A., ALVISI, L., AND DAHLIN, M. Separating agreement from execution for byzantine fault tolerant services. In *19th SOSP* (2003), pp. 253–267.

Case Study
GFS: Evolution on Fast-forward

A discussion between Kirk McKusick and Sean Quinlan about the origin and evolution of the Google File System.

During the early stages of development at Google, the initial thinking did not include plans for building a new file system. While work was still being done on one of the earliest versions of the company's crawl and indexing system, however, it became quite clear to the core engineers that they really had no other choice, and GFS (Google File System) was born.

First, given that Google's goal was to build a vast storage network out of inexpensive commodity hardware, it had to be assumed that component failures would be the norm—meaning that constant monitoring, error detection, fault tolerance, and automatic recovery would have to be an integral part of the file system. Also, even by Google's earliest estimates, the system's throughput requirements were going to be daunting by anybody's standards—featuring multi-gigabyte files and data sets containing terabytes of information and millions of objects. Clearly, this meant traditional assumptions about I/O operations and block sizes would have to be revisited. There was also the matter of scalability. This was a file system that would surely need to scale like no other. Of course, back in those earliest days, no one could have possibly imagined just how much scalability would be required. They would learn about that soon enough.

Still, nearly a decade later, most of Google's mind-boggling store of data and its ever-growing array of applications continue to rely upon GFS. Many adjustments have been made to the file system along the way, and—together with a fair number of accommodations implemented within the applications that use GFS—they have made the journey possible.

To explore the reasoning behind a few of the more crucial initial design decisions as well as some of the incremental adaptations that have been made since then, ACM asked Sean Quinlan to pull back the covers on the changing file-system requirements and the evolving thinking at Google. Since Quinlan served as the GFS tech leader for a couple of years and continues now as a principal engineer at Google, he's in a good position to offer that perspective. As a grounding point beyond the Googleplex, ACM asked Kirk McKusick to lead the discussion. He is best known for his work on BSD (Berkeley Software Distribution) Unix, including the original design of the Berkeley FFS (Fast File System).

The discussion starts, appropriately enough, at the beginning—with the unorthodox decision to base the initial GFS implementation on a single-master design. At first blush, the risk of a single centralized master becoming a bandwidth bottleneck—or, worse, a single point of failure—seems fairly obvious, but it turns out Google's engineers had their reasons for making this choice.

MCKUSICK One of the more interesting—and significant—aspects of the original GFS architecture was the decision to base it on a single master. Can you walk us through what led to that decision?
QUINLAN The decision to go with a single master was actually one of the very first decisions, mostly just to simplify the overall design problem. That is, building a distributed master right from the outset was deemed too difficult and would take too much time. Also, by going with the single-master

approach, the engineers were able to simplify a lot of problems. Having a central place to control replication and garbage collection and many other activities was definitely simpler than handling it all on a distributed basis. So the decision was made to centralize that in one machine.

MCKUSICK Was this mostly about being able to roll out something within a reasonably short time frame?

QUINLAN Yes. In fact, some of the engineers who were involved in that early effort later went on to build BigTable, a distributed storage system, but that effort took many years. The decision to build the original GFS around the single master really helped get something out into the hands of users much more rapidly than would have otherwise been possible.

Also, in sketching out the use cases they anticipated, it didn't seem the single-master design would cause much of a problem. The scale they were thinking about back then was framed in terms of hundreds of terabytes and a few million files. In fact, the system worked just fine to start with.

MCKUSICK But then what?

QUINLAN Problems started to occur once the size of the underlying storage increased. Going from a few hundred terabytes up to petabytes, and then up to tens of petabytes… that really required a proportionate increase in the amount of metadata the master had to maintain. Also, operations such as scanning the metadata to look for recoveries all scaled linearly with the volume of data. So the amount of work required of the master grew substantially. The amount of storage needed to retain all that information grew as well.

In addition, this proved to be a bottleneck for the clients, even though the clients issue few metadata operations themselves—for example, a client talks to the master whenever it does an open. When you have thousands of clients all talking to the master at the same time, given that the master is capable of doing only a few thousand operations a second, the average client isn't able to command all that many operations per second. Also bear in mind that there are applications such as MapReduce, where you might suddenly have a thousand tasks, each wanting to open a number of files. Obviously, it would take a long time to handle all those requests, and the master would be under a fair amount of duress.

MCKUSICK Now, under the current schema for GFS, you have one master per cell, right?

QUINLAN That's correct.

MCKUSICK And historically you've had one cell per data center, right?

QUINLAN That was initially the goal, but it didn't work out like that to a large extent—partly because of the limitations of the single-master design and partly because isolation proved to be difficult. As a consequence, people generally ended up with more than one cell per data center. We also ended up doing what we call a "multi-cell" approach, which basically made it possible to put multiple GFS masters on top of a pool of chunkservers. That way, the chunkservers could be configured to have, say, eight GFS masters assigned to them, and that would give you at least one pool of underlying storage—with multiple master heads on it, if you will. Then the application was responsible for partitioning data across those different cells.

MCKUSICK Presumably each application would then essentially have its own master that would be responsible for managing its own little file system. Was that basically the idea?

QUINLAN Well, yes and no. Applications would tend to use either one master or a small set of the masters. We also have something we called Name Spaces, which are just a very static way of partitioning a namespace that people can use to hide all of this from the actual application. The

Logs Processing System offers an example of this approach: once logs exhaust their ability to use just one cell, they move to multiple GFS cells; a namespace file describes how the log data is partitioned across those different cells and basically serves to hide the exact partitioning from the application. But this is all fairly static.

MCKUSICK What's the performance like, in light of all that?

QUINLAN We ended up putting a fair amount of effort into tuning master performance, and it's atypical of Google to put a lot of work into tuning any one particular binary. Generally, our approach is just to get things working reasonably well and then turn our focus to scalability—which usually works well in that you can generally get your performance back by scaling things. Because in this instance we had a single bottleneck that was starting to have an impact on operations, however, we felt that investing a bit of additional effort into making the master lighter weight would be really worthwhile. In the course of scaling from thousands of operations to tens of thousands and beyond, the single master had become somewhat less of a bottleneck. That was a case where paying more attention to the efficiency of that one binary definitely helped keep GFS going for quite a bit longer than would have otherwise been possible.

It could be argued that managing to get GFS ready for production in record time constituted a victory in its own right and that, by speeding Google to market, this ultimately contributed mightily to the company's success. A team of three was responsible for all of that—for the core of GFS—and for the system being readied for deployment in less than a year.

But then came the price that so often befalls any successful system—that is, once the scale and use cases have had time to expand far beyond what anyone could have possibly imagined. In Google's case, those pressures proved to be particularly intense.

Although organizations don't make a habit of exchanging file-system statistics, it's safe to assume that GFS is the largest file system in operation (in fact, that was probably true even before Google's acquisition of YouTube). Hence, even though the original architects of GFS felt they had provided adequately for at least a couple of orders of magnitude of growth, Google quickly zoomed right past that.

In addition, the number of applications GFS was called upon to support soon ballooned. In an interview with one of the original GFS architects, Howard Gobioff (conducted just prior to his surprising death in early 2008), he recalled, "The original consumer of all our earliest GFS versions was basically this tremendously large crawling and indexing system. The second wave came when our quality team and research groups started using GFS rather aggressively—and basically, they were all looking to use GFS to store large data sets. And then, before long, we had 50 users, all of whom required a little support from time to time so they'd all keep playing nicely with each other."

One thing that helped tremendously was that Google built not only the file system but also all of the applications running on top of it. While adjustments were continually made in GFS to make it more accommodating to all the new use cases, the applications themselves were also developed with the various strengths and weaknesses of GFS in mind. "Because we built everything, we were free to cheat whenever we wanted to," Gobioff neatly summarized. "We could push problems back and forth between the application space and the file-system space, and then work out accommodations between the two."

The matter of sheer scale, however, called for some more substantial adjustments. One coping strategy had to do with the use of multiple "cells" across the network, functioning essentially as related but distinct file systems. Besides helping to deal with the immediate problem of scale, this proved to be a more efficient arrangement for the operations of widely dispersed data centers.

Rapid growth also put pressure on another key parameter of the original GFS design: the choice to establish 64 MB as the standard chunk size. That, of course, was much larger than the typical file-system block size, but only because the files generated by Google's crawling and indexing system were unusually large. As the application mix changed over time, however, ways had to be found to let the system deal efficiently with large numbers of files requiring far less than 64 MB (think in terms of Gmail, for example). The problem was not so much with the number of files itself, but rather with the memory demands all of those files made on the centralized master, thus exposing one of the bottleneck risks inherent in the original GFS design.

MCKUSICK I gather from the original GFS paper [Ghemawat, S., Gobioff, H., Leung, S-T. 2003. The Google File System. SOSP (ACM Symposium on Operating Systems Principles)] that file counts have been a significant issue for you right along. Can you go into that a little bit?

QUINLAN The file-count issue came up fairly early because of the way people ended up designing their systems around GFS. Let me cite a specific example. Early in my time at Google, I was involved in the design of the Logs Processing system. We initially had a model where a front-end server would write a log, which we would then basically copy into GFS for processing and archival. That was fine to start with, but then the number of front-end servers increased, each rolling logs every day. At the same time, the number of log types was going up, and then you'd have front-end servers that would go through crash loops and generate lots more logs. So we ended up with a lot more files than we had anticipated based on our initial back-of-the-envelope estimates.

This became an area we really had to keep an eye on. Finally, we just had to concede there was no way we were going to survive a continuation of the sort of file-count growth we had been experiencing.

MCKUSICK Let me make sure I'm following this correctly: your issue with file-count growth is a result of your needing to have a piece of metadata on the master for each file, and that metadata has to fit in the master's memory.

QUINLAN That's correct.

MCKUSICK And there are only a finite number of files you can accommodate before the master runs out of memory?

QUINLAN Exactly. And there are two bits of metadata. One identifies the file, and the other points out the chunks that back that file. If you had a chunk that contained only 1 MB, it would take up only 1 MB of disk space, but it still would require those two bits of metadata on the master. If your average file size ends up dipping below 64 MB, the ratio of the number of objects on your master to what you have in storage starts to go down. That's where you run into problems.

Going back to that logs example, it quickly became apparent that the natural mapping we had thought of—and which seemed to make perfect sense back when we were doing our back-of-the-envelope estimates—turned out not to be acceptable at all. We needed to find a way to work around this by figuring out how we could combine some number of underlying objects into larger files. In the case of the logs, that wasn't exactly rocket science, but it did require a lot of effort.

MCKUSICK That sounds like the old days when IBM had only a minimum disk allocation, so it provided you with a utility that let you pack a bunch of files together and then create a table of contents for that.
QUINLAN Exactly. For us, each application essentially ended up doing that to varying degrees. That proved to be less burdensome for some applications than for others. In the case of our logs, we hadn't really been planning to delete individual log files. It was more likely that we would end up rewriting the logs to anonymize them or do something else along those lines. That way, you don't get the garbage-collection problems that can come up if you delete only some of the files within a bundle.

For some other applications, however, the file-count problem was more acute. Many times, the most natural design for some application just wouldn't fit into GFS—even though at first glance you would think the file count would be perfectly acceptable, it would turn out to be a problem. When we started using more shared cells, we put quotas on both file counts and storage space. The limit that people have ended up running into most has been, by far, the file-count quota. In comparison, the underlying storage quota rarely proves to be a problem.
MCKUSICK What longer-term strategy have you come up with for dealing with the file-count issue? Certainly, it doesn't seem that a distributed master is really going to help with that—not if the master still has to keep all the metadata in memory, that is.
QUINLAN The distributed master certainly allows you to grow file counts, in line with the number of machines you're willing to throw at it. That certainly helps.

One of the appeals of the distributed multimaster model is that if you scale everything up by two orders of magnitude, then getting down to a 1-MB average file size is going to be a lot different from having a 64-MB average file size. If you end up going below 1 MB, then you're also going to run into other issues that you really need to be careful about. For example, if you end up having to read 10,000 10-KB files, you're going to be doing a lot more seeking than if you're just reading 100 1-MB files.

My gut feeling is that if you design for an average 1-MB file size, then that should provide for a much larger class of things than does a design that assumes a 64-MB average file size. Ideally, you would like to imagine a system that goes all the way down to much smaller file sizes, but 1 MB seems a reasonable compromise in our environment.
MCKUSICK What have you been doing to design GFS to work with 1-MB files?
QUINLAN We haven't been doing anything with the existing GFS design. Our distributed master system that will provide for 1-MB files is essentially a whole new design. That way, we can aim for something on the order of 100 million files per master. You can also have hundreds of masters.
MCKUSICK So, essentially no single master would have all this data on it?
QUINLAN That's the idea.

With the recent emergence within Google of BigTable, a distributed storage system for managing structured data, one potential remedy for the file-count problem—albeit perhaps not the very best one—is now available.

The significance of BigTable goes far beyond file counts, however. Specifically, it was designed to scale into the petabyte range across hundreds or thousands of machines, as well as to make it easy to add more machines to the system and automatically start taking advantage of those resources without reconfiguration. For a company predicated on the notion of employing the collective power, potential redundancy, and economies of scale inherent in a massive deployment of commodity hardware, these rate as significant advantages indeed.

Accordingly, BigTable is now used in conjunction with a growing number of Google applications. Although it represents a departure of sorts from the past, it also must be said that BigTable was built on GFS, runs on GFS, and was consciously designed to remain consistent with most GFS principles. Consider it, therefore, as one of the major adaptations made along the way to help keep GFS viable in the face of rapid and widespread change.

MCKUSICK You now have this thing called BigTable. Do you view that as an application in its own right?

QUINLAN From the GFS point of view, it's an application, but it's clearly more of an infrastructure piece.

MCKUSICK If I understand this correctly, BigTable is essentially a lightweight relational database.

QUINLAN It's not really a relational database. I mean, we're not doing SQL and it doesn't really support joins and such. But BigTable is a structured storage system that lets you have lots of key-value pairs and a schema.

MCKUSICK Who are the real clients of BigTable?

QUINLAN BigTable is increasingly being used within Google for crawling and indexing systems, and we use it a lot within many of our client-facing applications. The truth of the matter is that there are tons of BigTable clients. Basically, any app with lots of small data items tends to use BigTable. That's especially true wherever there's fairly structured data.

MCKUSICK I guess the question I'm really trying to pose here is: Did BigTable just get stuck into a lot of these applications as an attempt to deal with the small-file problem, basically by taking a whole bunch of small things and then aggregating them together?

QUINLAN That has certainly been one use case for BigTable, but it was actually intended for a much more general sort of problem. If you're using BigTable in that way—that is, as a way of fighting the file-count problem where you might have otherwise used a file system to handle that—then you would not end up employing all of BigTable's functionality by any means. BigTable isn't really ideal for that purpose in that it requires resources for its own operations that are nontrivial. Also, it has a garbage-collection policy that's not super-aggressive, so that might not be the most efficient way to use your space. I'd say that the people who have been using BigTable purely to deal with the file-count problem probably haven't been terribly happy, but there's no question that it is one way for people to handle that problem.

MCKUSICK What I've read about GFS seems to suggest that the idea was to have only two basic data structures: logs and SSTables (Sorted String Tables). Since I'm guessing the SSTables must be used to handle key-value pairs and that sort of thing, how is that different from BigTable?

QUINLAN The main difference is that SSTables are immutable, while BigTable provides mutable key value storage, and a whole lot more. BigTable itself is actually built on top of logs and SSTables. Initially, it stores incoming data into transaction log files. Then it gets *compacted*—as we call it—into a series of SSTables, which in turn get compacted together over time. In some respects, it's reminiscent of a log-structure file system. Anyway, as you've observed, logs and SSTables do seem to be the two data structures underlying the way we structure most of our data. We have log files for mutable stuff as it's being recorded. Then, once you have enough of that, you sort it and put it into this structure that has an index.

Even though GFS does not provide a Posix interface, it still has a pretty generic file-system interface, so people are essentially free to write any sort of data they like. It's just that, over time, the majority of our users have ended up using these two data structures. We also have something called *protocol buffers*, which is our data description language. The majority of data ends up being protocol buffers in these two structures.

Both provide for compression and checksums. Even though there are some people internally who end up reinventing these things, most people are content just to use those two basic building blocks.

Because GFS was designed initially to enable a crawling and indexing system, throughput was everything. In fact, the original paper written about the system makes this quite explicit: "High sustained bandwidth is more important than low latency. Most of our target applications place a premium on processing data in bulk at a high rate, while few have stringent response-time requirements for an individual read and write."

But then Google either developed or embraced many user-facing Internet services for which this is most definitely not the case.

One GFS shortcoming that this immediately exposed had to do with the original single-master design. A single point of failure may not have been a disaster for batch-oriented applications, but it was certainly unacceptable for latency-sensitive applications, such as video serving. The later addition of automated failover capabilities helped, but even then service could be out for up to a minute.

The other major challenge for GFS, of course, has revolved around finding ways to build latency-sensitive applications on top of a file system designed around an entirely different set of priorities.

MCKUSICK It's well documented that the initial emphasis in designing GFS was on batch efficiency as opposed to low latency. Now that has come back to cause you trouble, particularly in terms of handling things such as videos. How are you handling that?

QUINLAN The GFS design model from the get-go was all about achieving throughput, not about the latency at which that might be achieved. To give you a concrete example, if you're writing a file, it will typically be written in triplicate—meaning you'll actually be writing to three chunkservers. Should one of those chunkservers die or hiccup for a long period of time, the GFS master will notice the problem and schedule what we call a *pullchunk*, which means it will basically replicate one of those chunks. That will get you back up to three copies, and then the system will pass control back to the client, which will continue writing.

When we do a pullchunk we limit it to something on the order of 5-10 MB a second. So, for 64 MB, you're talking about 10 seconds for this recovery to take place. There are lots of other things like this that might take 10 seconds to a minute, which works just fine for batch-type operations. If you're doing a large MapReduce operation, you're OK just so long as one of the items is not a real straggler, in which case you've got yourself a different sort of problem. Still, generally speaking, a hiccup on the order of a minute over the course of an hour-long batch job doesn't really show up. If you are working on Gmail, however, and you're trying to write a mutation that represents some user action, then getting stuck for a minute is really going to mess you up.

We've had similar issues with our master failover. Initially, GFS had no provision for automatic master failover. It was a manual process. Although it didn't happen a lot, whenever it did, the cell might be down for an hour. Even our initial master-failover implementation required on the order of minutes. Over the past year, however, we've taken that down to something on the order of tens of seconds.

MCKUSICK Still, for user-facing applications, that's not acceptable.

QUINLAN Right. While these instances—where you have to provide for failover and error recovery—may have been acceptable in the batch situation, they're definitely not OK from a latency point of view for a user-facing application. Another issue here is that there are places in the design where we've tried to optimize for throughput by dumping thousands of operations into a queue and then just processing through them. That leads to fine throughput, but it's not great for latency. You can easily get into situations where you might be stuck for seconds at a time in a queue just waiting to get to the head of the queue.

Our user base has definitely migrated from being a MapReduce-based world to more of an interactive world that relies on things such as BigTable. Gmail is an obvious example of that. Videos aren't quite as bad where GFS is concerned because you get to stream data, meaning you can buffer. Still, trying to build an interactive database on top of a file system that was designed from the start to support more batch-oriented operations has certainly proved to be a pain point.

MCKUSICK How exactly have you managed to deal with that?

QUINLAN Within GFS, we've managed to improve things to a certain degree, mostly by designing the applications to deal with the problems that come up. Take BigTable as a good concrete example. The BigTable transaction log is actually the biggest bottleneck for getting a transaction logged. In effect, we decided, "Well, we're going to see hiccups in these writes, so what we'll do is to have two logs open at any one time. Then we'll just basically merge the two. We'll write to one and if that gets stuck, we'll write to the other. We'll merge those logs once we do a replay—if we need to do a replay, that is." We tended to design our applications to function like that—which is to say they basically try to hide that latency since they know the system underneath isn't really all that great.

The guys who built Gmail went to a multihomed model, so if one instance of your Gmail account got stuck, you would basically just get moved to another data center. Actually, that capability was needed anyway just to ensure availability. Still, part of the motivation was that they wanted to hide the GFS problems.

MCKUSICK I think it's fair to say that, by moving to a distributed-master file system, you're definitely going to be able to attack some of those latency issues.

QUINLAN That was certainly one of our design goals. Also, BigTable itself is a very failure-aware system that tries to respond to failures far more rapidly than we were able to before. Using that as our metadata storage helps with some of those latency issues as well.

The engineers who worked on the earliest versions of GFS weren't particularly shy about departing from traditional choices in file-system design whenever they felt the need to do so. It just so happens that the approach taken to consistency is one of the aspects of the system where this is particularly evident.

Part of this, of course, was driven by necessity. Since Google's plans rested largely on massive deployments of commodity hardware, failures and hardware-related faults were a given. Beyond that, according to the original GFS paper, there were a few compatibility issues. "Many of our disks claimed to the Linux driver that they supported a range of IDE protocol versions but in fact responded reliably only to the more recent ones. Since the protocol versions are very similar, these drives mostly worked but occasionally the mismatches would cause the drive and the kernel to disagree about the drive's state. This would corrupt data silently due to problems in the kernel. This problem motivated our use of checksums to detect data corruption."

That didn't mean just any checksumming, however, but instead rigorous end-to-end checksumming, with an eye to everything from disk corruption to TCP/IP corruption to machine backplane corruption.

Interestingly, for all that checksumming vigilance, the GFS engineering team also opted for an approach to consistency that's relatively loose by file-system standards. Basically, GFS simply accepts that there will be times when people will end up reading slightly stale data. Since GFS is used mostly as an append-only system as opposed to an overwriting system, this generally means those people might end up missing something that was appended to the end of the file after they'd already opened it. To the GFS designers, this seemed an acceptable cost (although it turns out that there are applications for which this proves problematic).

Also, as Gobioff explained, "The risk of stale data in certain circumstances is just inherent to a highly distributed architecture that doesn't ask the master to maintain all that much information. We definitely could have made things a lot tighter if we were willing to dump a lot more data into the master and then have it maintain more state. But that just really wasn't all that critical to us."

Perhaps an even more important issue here is that the engineers making this decision owned not just the file system but also the applications intended to run on the file system. According to Gobioff, "The thing is that we controlled both the horizontal and the vertical—the file system and the application. So we could be sure our applications would know what to expect from the file system. And we just decided to push some of the complexity out to the applications to let them deal with it."

Still, there are some at Google who wonder whether that was the right call if only because people can sometimes obtain different data in the course of reading a given file multiple times, which tends to be so strongly at odds with their whole notion of how data storage is supposed to work.

MCKUSICK Let's talk about consistency. The issue seems to be that it presumably takes some amount of time to get everything fully written to all the replicas. I think you said something earlier to the effect that GFS essentially requires that this all be fully written before you can continue.
QUINLAN That's correct.
MCKUSICK If that's the case, then how can you possibly end up with things that aren't consistent?
QUINLAN Client failures have a way of fouling things up. Basically, the model in GFS is that the client just continues to push the write until it succeeds. If the client ends up crashing in the middle of an operation, things are left in a bit of an indeterminate state.

Early on, that was sort of considered to be OK, but over time, we tightened the window for how long that inconsistency could be tolerated, and then we slowly continued to reduce that. Otherwise, whenever the data is in that inconsistent state, you may get different lengths for the file. That can lead to some confusion. We had to have some backdoor interfaces for checking the consistency of the file data in those instances. We also have something called RecordAppend, which is an interface

designed for multiple writers to append to a log concurrently. There the consistency was designed to be very loose. In retrospect, that turned out to be a lot more painful than anyone expected.
MCKUSICK What exactly was loose? If the primary replica picks what the offset is for each write and then makes sure that actually occurs, I don't see where the inconsistencies are going to come up.
QUINLAN What happens is that the primary will try. It will pick an offset, it will do the writes, but then one of them won't actually get written. Then the primary might change, at which point it can pick a different offset. RecordAppend does not offer any replay protection either. You could end up getting the data multiple times in the file.

There were even situations where you could get the data in a different order. It might appear multiple times in one chunk replica, but not necessarily in all of them. If you were reading the file, you could discover the data in different ways at different times. At the record level, you could discover the records in different orders depending on which chunks you happened to be reading.
MCKUSICK Was this done by design?
QUINLAN At the time, it must have seemed like a good idea, but in retrospect I think the consensus is that it proved to be more painful than it was worth. It just doesn't meet the expectations people have of a file system, so they end up getting surprised. Then they had to figure out work-arounds.
MCKUSICK In retrospect, how would you handle this differently?
QUINLAN I think it makes more sense to have a single writer per file.
MCKUSICK All right, but what happens when you have multiple people wanting to append to a log?
QUINLAN You serialize the writes through a single process that can ensure the replicas are consistent.
MCKUSICK There's also this business where you essentially snapshot a chunk. Presumably, that's something you use when you're essentially replacing a replica, or whenever some chunkserver goes down and you need to replace some of its files.
QUINLAN Actually, two things are going on there. One, as you suggest, is the recovery mechanism, which definitely involves copying around replicas of the file. The way that works in GFS is that we basically revoke the lock so that the client can't write it anymore, and this is part of that latency issue we were talking about.

There's also a separate issue, which is to support the snapshot feature of GFS. GFS has the most general-purpose snapshot capability you can imagine. You could snapshot any directory somewhere, and then both copies would be entirely equivalent. They would share the unchanged data. You could change either one and you could further snapshot either one. So it was really more of a clone than what most people think of as a snapshot. It's an interesting thing, but it makes for difficulties—especially as you try to build more distributed systems and you want potentially to snapshot larger chunks of the file tree.

I also think it's interesting that the snapshot feature hasn't been used more since it's actually a very powerful feature. That is, from a file-system point of view, it really offers a pretty nice piece of functionality. But putting snapshots into file systems, as I'm sure you know, is a real pain.
MCKUSICK: I know. I've done it. It's excruciating—especially in an overwriting file system.
QUINLAN Exactly. This is a case where we didn't cheat, but from an implementation perspective, it's hard to create true snapshots. Still, it seems that in this case, going the full deal was the right decision. Just the same, it's an interesting contrast to some of the other decisions that were made early on in terms of the semantics.

All in all, the report card on GFS nearly 10 years later seems positive. There have been problems and shortcomings, to be sure, but there's surely no arguing with Google's success and GFS has without a doubt played an important role in that. What's more, its staying power has been nothing short of remarkable given that Google's operations have scaled orders of magnitude beyond anything the system had been designed to handle, while the application mix Google currently supports is not one that anyone could have possibly imagined back in the late '90s.

Still, there's no question that GFS faces many challenges now. For one thing, the awkwardness of supporting an ever-growing fleet of user-facing, latency-sensitive applications on top of a system initially designed for batch-system throughput is something that's obvious to all.

The advent of BigTable has helped somewhat in this regard. As it turns out, however, BigTable isn't actually all that great a fit for GFS. In fact, it just makes the bottleneck limitations of the system's single-master design more apparent than would otherwise be the case.

For these and other reasons, engineers at Google have been working for much of the past two years on a new distributed master system designed to take full advantage of BigTable to attack some of those problems that have proved particularly difficult for GFS.

Accordingly, it now seems that beyond all the adjustments made to ensure the continued survival of GFS, the newest branch on the evolutionary tree will continue to grow in significance over the years to come. Q

LOVE IT, HATE IT? LET US KNOW
feedback@queue.acm.org

Interpreting the Data:
Parallel Analysis with Sawzall

Rob Pike, Sean Dorward, Robert Griesemer, Sean Quinlan
Google, Inc.

Abstract

Very large data sets often have a flat but regular structure and span multiple disks and machines. Examples include telephone call records, network logs, and web document repositories. These large data sets are not amenable to study using traditional database techniques, if only because they can be too large to fit in a single relational database. On the other hand, many of the analyses done on them can be expressed using simple, easily distributed computations: filtering, aggregation, extraction of statistics, and so on.

We present a system for automating such analyses. A filtering phase, in which a query is expressed using a new procedural programming language, emits data to an aggregation phase. Both phases are distributed over hundreds or even thousands of computers. The results are then collated and saved to a file. The design—including the separation into two phases, the form of the programming language, and the properties of the aggregators—exploits the parallelism inherent in having data and computation distributed across many machines.

1 Introduction

Many data sets are too large, too dynamic, or just too unwieldy to be housed productively in a relational database. One common scenario is a set of many plain files—sometimes amounting to petabytes of data—distributed across many disks on many computers (Figure 1). The files in turn comprise many records, organized along some obvious axes such as time or geography. Examples might include a web page repository used to construct the index for an internet search engine, the system health records from thousands of on-line server machines, telephone call records or other business transaction logs, network packet traces, web server query logs, or even higher-level data such as satellite imagery.

Quite often the analyses applied to these data sets can be expressed simply, using operations much less sophisticated than a general SQL query. For instance, we might wish to count records that satisfy a certain property, or extract them, or search for anomalous records, or construct frequency histograms of the values of certain fields in the records. In other cases the analyses might be more intricate but still be expressible as a series of simpler steps, operations that can be mapped easily onto a set of records in files.

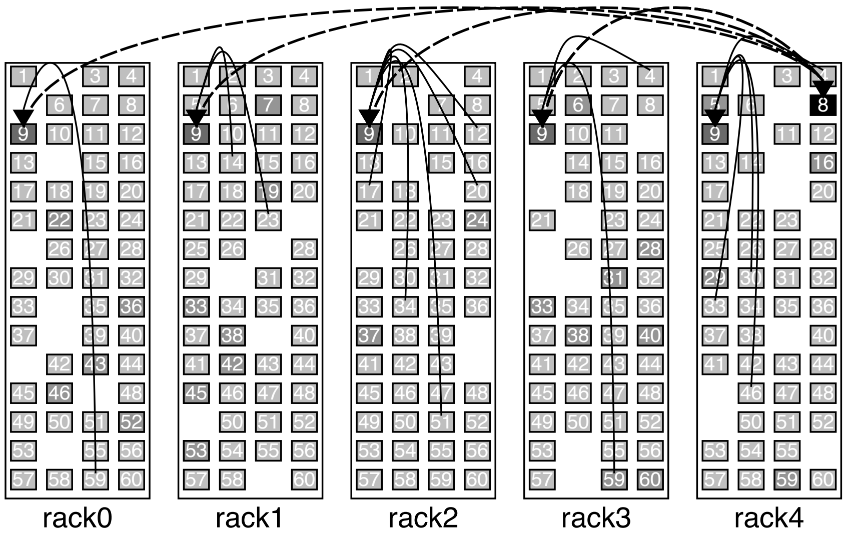


Figure 1: Five racks of 50-55 working computers each, with four disks per machine. Such a configuration might have a hundred terabytes of data to be processed, distributed across some or all of the machines. Tremendous parallelism can be achieved by running a filtering phase independently on all 250+ machines and aggregating their emitted results over a network between them (the arcs). Solid arcs represent data flowing from the analysis machines to the aggregators; dashed arcs represent the aggregated data being merged, first into one file per aggregation machine and then to a single final, collated output file.

Not all problems are like this, of course. Some benefit from the tools of traditional programming—arithmetic, patterns, storage of intermediate values, functions, and so on—provided by procedural languages such as Python [1] or Awk [12]. If the data is unstructured or textual, or if the query requires extensive calculation to arrive at the relevant data for each record, or if many distinct but related calculations must be applied during a single scan of the data set, a procedural language is often the right tool. Awk was specifically designed for making data mining easy. Although it is used for many other things today, it was inspired by a tool written by Marc Rochkind that executed procedural code when a regular expression matched a record in telephone system log data [13]. Even today, there are large Awk programs in use for mining telephone logs. Languages such as C or C++, while capable of handling such tasks, are more awkward to use and require more effort on the part of the programmer. Still, Awk and Python are not panaceas; for instance, they have no inherent facilities for processing data on multiple machines.

Since the data records we wish to process do live on many machines, it would be fruitful to exploit the combined computing power to perform these analyses. In particular, if the individual steps can be expressed as query operations that can be evaluated one record at a time, we can distribute the calculation across all the machines and achieve very high throughput. The results of these operations will then require an aggregation phase. For example, if we are counting records, we need to gather the counts from the individual machines before we can report the total count.

We therefore break our calculations into two phases. The first phase evaluates the analysis on each record individually, while the second phase aggregates the results (Figure 2). The system described in this paper goes even further, however. The analysis in the first phase is expressed in a new procedural programming language that executes one record at a time, in isolation, to calculate query results for each record. The second phase is restricted to a set of predefined aggregators that process the intermediate results generated by the first phase. By restricting the calculations to this model, we can achieve very high throughput. Although not all calculations fit this model well, the ability to harness a thousand or more machines with a few lines of code provides some compensation.

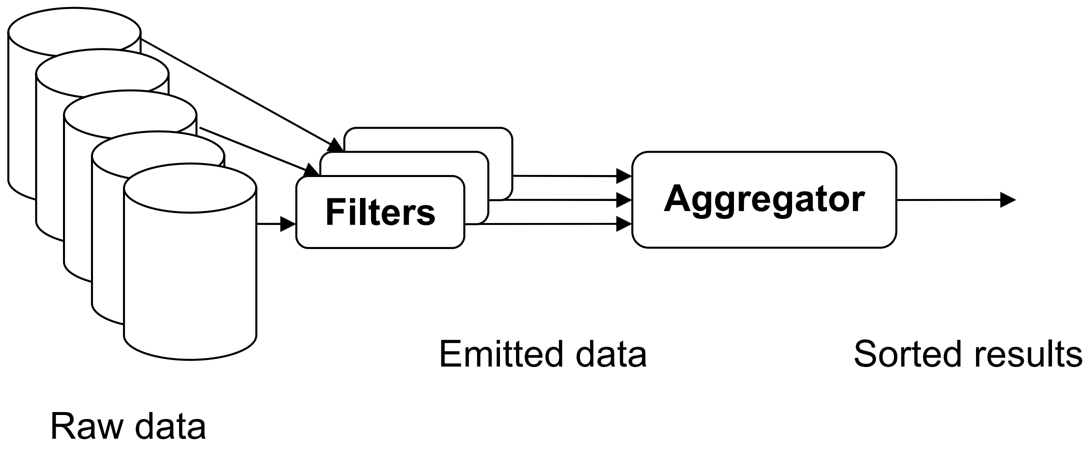


Figure 2: The overall flow of filtering, aggregating, and collating. Each stage typically involves less data than the previous.

Of course, there are still many subproblems that remain to be solved. The calculation must be divided into pieces and distributed across the machines holding the data, keeping the computation as near the data as possible to avoid network bottlenecks. And when there are many machines there is a high probability of some of them failing during the analysis, so the system must be fault tolerant. These are difficult and interesting problems, but they should be handled without the involvement of the user of the system. Google has several pieces of infrastructure, including GFS [9] and MapReduce [8], that cope with fault tolerance and reliability and provide a powerful framework upon which to implement a large, parallel system for distributed analysis. We therefore hide the details from the user and concentrate on the job at hand: expressing the analysis cleanly and executing it quickly.

2 Overview

A typical job will process anywhere from a gigabyte to many terabytes of data on hundreds or even thousands of machines in parallel, some executing the query while others aggregate the results. An analysis may consume months of CPU time, but with a thousand machines that will only take a few hours of real time.

Our system's design is influenced by two observations.

First, if the querying operations are commutative across records, the order in which the records are processed is unimportant. We can therefore work through the input in arbitrary order.

Second, if the aggregation operations are commutative, the order in which the intermediate values are processed is unimportant. Moreover, if they are also associative, the intermediate values can be grouped arbitrarily or even aggregated in stages. As an example, counting involves addition, which is guaranteed to be the same independent of the order in which the values are added and independent of any intermediate subtotals that are formed to coalesce intermediate values.

The constraints of commutativity and associativity are not too restrictive; they permit a wide range of valuable analyses, including: counting, filtering, sampling, generating histograms, finding the most frequent items, and many more.

The set of aggregations is limited but the query phase can involve more general computations, which we express in a new interpreted, procedural programming language called Sawzall.[1] (An interpreted language is fast enough: most of the programs are small and on large data sets the calculation tends to be I/O bound, as is discussed in the section on performance.)

An analysis operates as follows. First the input is divided into pieces to be processed separately, perhaps individual files or groups of records, which are located on multiple storage nodes.

Next, a Sawzall interpreter is instantiated for each piece of data. This will involve many machines, perhaps the same machines that store the data or perhaps a different, nearby set. There will often be more data pieces than computers, in which case scheduling and load balancing software will dispatch the pieces to machines as they complete the processing of prior pieces.

[1]Not related to the portable reciprocating power saw trademark of the Milwaukee Electric Tool Corporation.

The Sawzall program operates on each input record individually. The output of the program is, for each record, zero or more intermediate values—integers, strings, key-value pairs, tuples, etc.—to be combined with values from other records.

These intermediate values are sent to further computation nodes running the aggregators, which collate and reduce the intermediate values to create the final results. In a typical run, the majority of machines will run Sawzall and a smaller fraction will run the aggregators, reflecting not only computational overhead but also network load balancing issues; at each stage, the amount of data flowing is less than at the stage before (see Figure 2).

Once all the processing is complete, the results will be spread across multiple files, one per aggregation machine. A final step collects, collates, and formats the final results, to be viewed or stored in a single file.

3 A Simple Example

These ideas may become clearer in the context of a simple example. Let's say our input is a set of files comprising records that each contain one floating-point number. This complete Sawzall program will read the input and produce three results: the number of records, the sum of the values, and the sum of the squares of the values.

```
count: table sum of int;
total: table sum of float;
sum_of_squares: table sum of float;
x: float = input;
emit count <- 1;
emit total <- x;
emit sum_of_squares <- x * x;
```

The first three lines declare the aggregators `count`, `total`, and `sum_of_squares`. The keyword `table` introduces an aggregator type; aggregators are called tables in Sawzall even though they may be singletons. These particular tables are `sum` tables; they add up the values emitted to them, `ints` or `floats` as appropriate.

For each input record, Sawzall initializes the pre-defined variable `input` to the uninterpreted byte string of the input record. Therefore, the line

```
x: float = input;
```

converts the input record from its external representation into a native floating-point number, which is stored in a local variable `x`. Finally, the three `emit` statements send intermediate values to the aggregators.

When run, the program is instantiated once for each input record. Local variables declared by the program are created anew for each instance, while tables declared in the program are shared by all instances. Emitted values are summed up by the global tables. When all records have been processed, the values in the tables are saved to one or more files.

The next few sections outline some of the Google infrastructure this system is built upon: protocol buffers, the Google File System, the Workqueue, and MapReduce. Subsequent sections describe the language and other novel components of the system in more detail.

4 Protocol Buffers

Although originally conceived for defining the messages communicated between servers, Google's *protocol buffers* are also used to describe the format of permanent records stored on disk. They serve a purpose similar to XML, but have a much denser, binary representation, which is in turn often wrapped by an additional compression layer for even greater density.

Protocol buffers are described by a data description language (DDL) that defines the content of the messages. A *protocol compiler* takes this language and generates executable code to manipulate the protocol buffers. A flag to the protocol compiler specifies the output language: C++, Java, Python, and so on. For each protocol buffer type in the DDL file, the output of the protocol compiler includes the native data structure definition, code to access the fields, marshaling and unmarshaling code to translate between the native representation and the external binary format, and debugging and pretty-printing interfaces. The generated code is compiled and linked with the application to provide efficient, clean access to the records. There is also a tool suite for examining and debugging stored protocol buffers.

The DDL forms a clear, compact, extensible notation describing the layout of the binary records and naming the fields. Protocol buffer types are roughly analogous to C `structs`: they comprise named, typed fields. However, the DDL includes two additional properties for each field: a distinguishing integral tag, used to identify the field in the binary representation, and an indication of whether the field is required or optional. These additional properties allow for backward-compatible extension of a protocol buffer, by marking all new fields as optional and assigning them an unused tag.

For example, the following describes a protocol buffer with two required fields. Field `x` has tag 1, and field `y` has tag 2.

```
parsed message Point {
    required int32 x = 1;
    required int32 y = 2;
};
```

To extend this two-dimensional point, one can add a new, optional field with a new tag. All existing Points stored on disk remain readable; they are compatible with the new definition since the new field is optional.

```
parsed message Point {
    required int32 x = 1;
    required int32 y = 2;
    optional string label = 3;
};
```

Most of the data sets our system operates on are stored as records in protocol buffer format. The protocol compiler was extended by adding support for Sawzall to enable convenient, efficient I/O of protocol buffers in the new language.

5 Google File System (GFS)

The data sets are often stored in GFS, the Google File System [9]. GFS provides a reliable distributed storage system that can grow to petabyte scale by keeping data in 64-megabyte "chunks" stored on disks spread across thousands of machines. Each chunk is replicated, usually 3 times, on different machines so GFS can recover seamlessly from disk or machine failure.

GFS runs as an application-level file system with a traditional hierarchical naming scheme. The data sets themselves have regular structure, represented by a large set of individual GFS files each around a gigabyte in size. For example, a document repository (the result of a web crawl) holding a few billion HTML pages might be stored as several thousand files each storing a million or so documents of a few kilobytes each, compressed.

6 Workqueue and MapReduce

The business of scheduling a job to run on a cluster of machines is handled by software called (somewhat misleadingly) the Workqueue. In effect, the Workqueue creates a large-scale time sharing system out of an array of computers and their disks. It schedules jobs, allocates resources, reports status, and collects the results.

The Workqueue is similar to several other systems such as Condor [16]. We often overlay a Workqueue cluster and a GFS cluster on the same set of machines. Since GFS is a storage system, its CPUs are often lightly loaded, and the free computing cycles can be used to run Workqueue jobs.

MapReduce [8] is a software library for applications that run on the Workqueue. It performs three primary services. First, it provides an execution model for programs that operate on many data items in parallel. Second, it isolates the application from the details of running a distributed program, including issues such as data distribution, scheduling, and fault tolerance. Finally, when possible it schedules the computations so each unit runs on the machine or rack that holds its GFS data, reducing the load on the network.

As the name implies, the execution model consists of two phases: a first phase that *maps* an execution across all the items in the data set; and a second phase that *reduces* the results of the first phase to arrive at a final answer. For example, a sort program using MapReduce would map a standard sort algorithm upon each of the files in the data set, then reduce the individual results by running a merge sort to produce the final output. On a thousand-machine cluster, a MapReduce implementation can sort a terabyte of data at an aggregate rate of a gigabyte per second [9].

Our data processing system is built on top of MapReduce. The Sawzall interpreter runs in the map phase. It is instantiated in parallel on many machines, with each instantiation processing one file or perhaps GFS chunk. The Sawzall program executes once for each record of the data set. The output of this map phase is a set of data items to be accumulated in the aggregators. The aggregators run in the reduce phase to condense the results to the final output.

The following sections describe these pieces in more detail.

7 Sawzall Language Overview

The query language, Sawzall, operates at about the level of a type-safe scripting language. For problems that can be solved in Sawzall, the resulting code is much simpler and shorter—by a factor of ten or more—than the corresponding C++ code in MapReduce.

The syntax of statements and expressions is borrowed largely from C; `for` loops, `while` loops, `if` statements and so on take their familiar form. Declarations borrow from the Pascal tradition:

```
i: int;       # a simple integer declaration
i: int = 0;   # a declaration with an initial value
```

The basic types include integer (`int`), a 64-bit signed value; floating point (`float`), a double-precision IEEE value; and special integer-like types called `time` and `fingerprint`, also 64 bits long. The `time` type has microsecond resolution and the libraries include convenient functions for decomposing and manipulating these values. The `fingerprint` type represents an internally-computed hash of another value, which makes it easy to construct data structures such as aggregators indexed by the fingerprint of a datum. As another example, one might use the fingerprint of a URL or of its HTML contents to do efficient comparison or fair selection among a set of documents.

There are also two primitive array-like types: `bytes`, similar to a C array of `unsigned char`; and `string`, which is defined to hold characters from the Unicode character set. There is no "character" type; the elements of byte arrays and strings are `int` values, although an `int` is capable of holding a much larger value than will fit in a byte or string element.

Compound types include arrays, maps (an overloaded term in this paper), and tuples. Arrays are indexed by integer values, while maps are like associative arrays or Python dictionaries and may be indexed by any type, with the indices unordered and the storage for the elements created on demand. Finally, tuples represent arbitrary groupings of data, like a C struct or Pascal record. A `typedef`-like mechanism allows any type to be given a shorthand name.

Conversion operators translate values from one type to another, and encapsulate a wide range of conversions. For instance, to extract the floating point number represented by a string, one simply converts the value:

```
f: float;
s: string = "1.234";
f = float(s);
```

Some conversions take parameters; for example

```
string(1234, 16)
```

generates the hexadecimal representation of the integer, while

```
string(utf8_bytes, "UTF-8")
```

will interpret the byte array in UTF-8 representation and return a string of the resulting Unicode character values.

For convenience, and to avoid the stutter endemic in some languages' declaration syntaxes, in initializations the appropriate conversion operation (with default parameter values) is supplied implicitly by the compiler. Thus

```
b: bytes = "Hello, world!\n";
```

is equivalent to the more verbose

```
b: bytes = bytes("Hello, world!\n", "UTF-8");
```

Values of any type can be converted into strings to aid debugging.

One of the most important conversion operations is associated with protocol buffers. A compile-time directive in Sawzall, `proto`, somewhat analogous to C's `#include` directive, imports the DDL for a protocol buffer from a file and defines the Sawzall tuple type that describes the layout. Given that tuple description, one can convert the input protocol buffer to a Sawzall value.

For each input record, the special variable `input` is initialized by the interpreter to the uninterpreted byte array holding the data, typically a protocol buffer. In effect, the execution of the Sawzall program for each record begins with the implicit statement:

```
input: bytes = next_record_from_input();
```

Therefore, if the file `some_record.proto` includes the definition of a protocol buffer of type `Record`, the following code will parse each input record and store it in the variable `r`:

```
proto "some_record.proto"  # define 'Record'
r: Record = input;         # convert input to Record
```

The language has a number of other traditional features such as functions and a wide selection of intrinsic library calls. Among the intrinsics are functions that connect to existing code for internationalization, document parsing, and so on.

7.1 Input and Aggregation

Although at the statement level Sawzall is a fairly ordinary language, it has two very unusual features, both in some sense outside the language itself:

1. A Sawzall program defines the operations to be performed on a single record of the data. There is nothing in the language to enable examining multiple input records simultaneously, or even to have the contents of one input record influence the processing of another.
2. The only output primitive in the language is the `emit` statement, which sends data to an external aggregator that gathers the results from each record and correlates and processes the result.

Thus the usual behavior of a Sawzall program is to take the `input` variable, parse it into a data structure by a conversion operation, examine the data, and emit some values. We saw this pattern in the simple example of Section 3.

Here is a more representative Sawzall program. Given a set of logs of the submissions to our source code management system, this program will show how the rate of submission varies through the week, at one-minute resolution:

```
proto "p4stat.proto"
submitsthroughweek: table sum[minute: int] of count: int;

log: P4ChangelistStats = input;
t: time = log.time;  # microseconds

minute: int = minuteof(t)+60*(hourof(t)+24*(dayofweek(t)-1));
```

```
    emit submitsthroughweek[minute] <- 1;
```

The program begins by importing the protocol buffer description from the file `p4stat.proto`. In particular, it declares the type `P4ChangelistStats`. (The programmer must know the type that will arise from the `proto` directive, but this is an accepted property of the protocol buffer DDL.)

The declaration of `submitsthroughweek` appears next. It defines a table of sum values, indexed by the integer minute. Note that the index value in the declaration of the table is given an optional name (`minute`). This name serves no semantic purpose, but makes the declaration easier to understand and provides a label for the field in the aggregated output.

The declaration of `log` converts the `input` byte array (using code generated by the `proto` directive) into the Sawzall type `P4ChangelistStats`, a tuple, and stores the result in the variable `log`. Then we pull out the time value and, for brevity, store it in the variable `t`.

The next declaration has a more complicated initialization expression that uses some built-in functions to extract the cardinal number of the minute of the week from the time value.

Finally, the `emit` statement counts the submission represented in this record by adding its contribution to the particular minute of the week in which it appeared.

To summarize, this program, for each record, extracts the time stamp, bins the time to a minute within the week, and adds one to the count for that minute. Then, implicitly, it starts again on the next record.

When we run this program over all the submission logs—which span many months—and plot the result, we see an aggregated view of how activity tends to vary through the week, minute by minute. The output looks like this:

```
  submitsthroughweek[0] = 27
  submitsthroughweek[1] = 31
  submitsthroughweek[2] = 52
  submitsthroughweek[3] = 41
  ...
  submitsthroughweek[10079] = 34
```

When plotted, the graph looks like Figure 3.

The salient point is not the data itself, of course, but the simplicity of the program that extracts it.

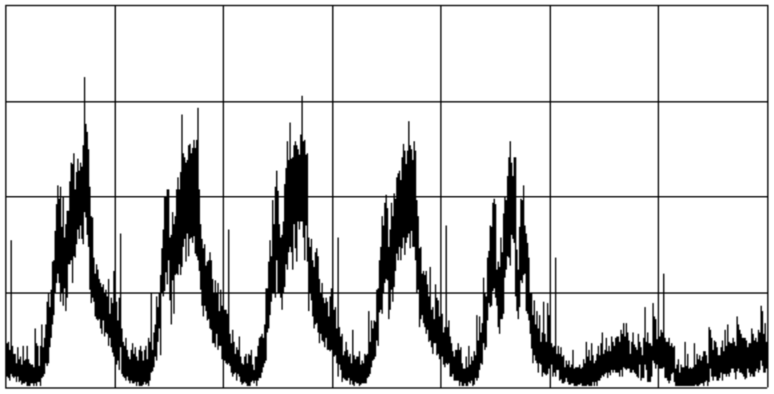


Figure 3: Frequency of submits to the source code repository through the week. The graph starts at midnight Monday morning.

7.2 More About Aggregators

Aggregation is done outside the language for a couple of reasons. A more traditional language would use the language itself to correlate results, but some of the aggregation algorithms are sophisticated and best implemented in a native language and packaged in some form. More important, drawing an explicit line between filtering and aggregation enables a high degree of parallelism, even though it hides the parallelism from the language itself. Nowhere in Sawzall is the multiplicity of records apparent, yet a typical Sawzall job operates on billions of records, often on hundreds or thousands of machines simultaneously.

Focusing attention on the aggregators encourages the creation of unusual statistical collectors. There are a number of aggregators available; here is an incomplete list, with examples:

- Collection:

  ```
  c: table collection of string;
  ```

  A simple list of all the emitted values, including all duplicates, in arbitrary order.

- Sample:

  ```
  s: table sample(100) of string;
  ```

  Like collection, but chooses an unbiased sample of the emitted values. The size of the desired sample is provided as a parameter.

- Sum:

  ```
  s: table sum of { count: int, revenue: float };
  ```

  The summation of all the emitted values. The values must be arithmetic or composed of arithmetic values, as is the tuple in the example. For compound values, the components are summed elementwise. In the example shown, if `count` were always 1 when emitted, the average revenue could be computed after the run by dividing `revenue` by `count`.

- Maximum:

  ```
  m: table maximum(10) of string weight length: int;
  ```

  The highest-weighted values. The values are tagged with the weight, and the value with the highest weight is chosen. The parameter (10 in the example) specifies the number of values to keep. The weight, introduced by the obvious keyword, has its type (here `int`) provided in the declaration and its value in the `emit` statement. For the example given,

  ```
  emit m <- s weight len(s);
  ```

  will report the ten longest strings in the data.

- Quantile:

  ```
  q: table quantile(101) of response_in_ms: int;
  ```

  Use the set of emitted values to construct a cumulative probability distribution represented by the quantile values for each increment of probability. (The algorithm is a distributed variant of that of Greenwald and Khanna [10].) The example could be used to see how system response varies. With parameter 101, it computes percentiles; table indices in that case range from 0 to 100, so the element at index 50 records the median time, while the element at index 99 is the time value for which 99% of the response times were equal or lower.

- Top:

  ```
  t: table top(10) of language: string;
  ```

  Estimate which values are the most popular. (By contrast, the maximum table finds the items with the highest weight, not the highest frequency of occurrence.)

  For our example,

  ```
  emit t <- language_of_document(input);
  ```

  will estimate the ten most frequently occurring languages found in a document repository.

  For large data sets, it can be prohibitively expensive to find the precise ordering of frequency of occurrence, but there are efficient estimators. The top table uses a distributed variant of the algorithm by Charikar, Chen, and Farach-Colton [5]. The algorithm is approximate: with high probability it returns approximately the correct top elements. Its commutativity and associativity are also approximate: changing the order that the input is processed can change the final results. To compensate, in addition to computing the counts of the elements, we compute the estimated error of those counts. If the error is small compared to the counts, the quality of the results is high, while if the error is relatively large, chances are the results are bad. The algorithm used in the top tables works well for skewed distributions but produces unreliable results when all the values occur with similar frequency. For our analyses this is rarely an issue.

- Unique:

  ```
  u: table unique(10000) of string;
  ```

  The unique table is unusual. It reports the estimated size of the population of unique items emitted to it. A sum table could be used to count the total number of elements, but a unique table will ignore duplicates; in effect it computes the size of the set of input values. The unique table is also unusual in that its output is always a count, regardless of the type of the values emitted to it. The parameter specifies the size of the internal table used to do the estimation; a value of 10000 generates a final value within $\pm 2\%$ of the correct answer with 95% probability. (For set size N, the standard deviation is about $N \times param^{-1/2}$.)

7.3 Implementing an Aggregator

Occasionally, ideas arise for new aggregators to be supported by Sawzall. Adding new aggregators is fairly easy, although not trivial. Their implementations are connected to the Sawzall runtime and interact with system internals, managing low-level data formats and distributed computations. Moreover, most aggregators handle many different Sawzall data types, further complicating their implementations. There is a support in the Sawzall runtime to ease these tasks, but we have not lavished these libraries with the same care given the Sawzall language.

The Sawzall runtime itself manages the plumbing of emitting values, passing data between machines, and so on, while a set of five functions specific to each aggregator class handles type checking, allocation of aggregation state, merging emitted values, packaging the data into an intermediate form suitable for further merging, and producing the final output data. To add a new aggregator, the programmer implements this five-function interface and links it with the Sawzall runtime. For simple aggregators, such as `sum`, the implementation is straightforward. For more sophisticated aggregators, such as `quantile` and `top`, care must be taken to choose an algorithm that is commutative, associative and reasonably efficient for distributed processing. Our smallest aggregators are implemented in about 200 lines of C++, while the largest require about 1000.

Some aggregators can process data as part of the mapping phase to reduce the network bandwidth to the aggregators. For instance, a `sum` table can add the individual elements locally, emitting only an occasional subtotal value to the remote aggregator. In MapReduce terminology, this is the *combining* phase of the MapReduce run, a sort of middle optimization step between map and reduce.

It would be possible to create a new aggregator-specification language, or perhaps extend Sawzall to handle the job. However, we have found that on the rare occasion that a new aggregator is needed, it's been easy to implement. Moreover, an aggregator's performance is crucial for the performance of the system, and since it needs to interact with the Mapreduce framework, it is best expressed in C++, the language of the Sawzall runtime and Mapreduce.

7.4 Indexed Aggregators

An aggregator can be indexed, which in effect creates a distinct individual aggregator for each unique value of the index. The index can be of any Sawzall type and can be compounded to construct multidimensional aggregators.

For example, if we are examining web server logs, the table

```
  table top(1000) [country: string][hour: int] of request: string;
```

could be used to find the 1000 most popular request strings for each country, for each hour.

The Sawzall runtime automatically creates individual aggregators as new index values arise, similarly to maps adding entries as new key values occur. The aggregation phase will collate values according to index and generate the appropriate aggregated values for each distinct index value.

As part of the collation, the values are sorted in index order, to facilitate merging values from different machines. When the job completes, the values are therefore arranged in index order, which means the output from the aggregators is in index order.

The indices themselves form a useful source of information. To return to the web server example above, after the run the set of values recorded in the `country` index form a record of the set of countries from which requests were received. In effect, the properties of index generation turn the indices into a way to recover sets of values. The indices resulting from

```
    t1: table sum[country: string] of int
```

would be equivalent to the values collected by

```
    t2: table collection of country: string
```

with duplicates removed. Its output would be of the form

```
    t1["china"] = 123456
    t1["japan"] = 142367
    ...
```

which the user would need to post-process with a separate tool to extract the set of countries.

8 System Model

Now that the basic features of the language have been presented, we can give an overview of the high-level system model by which data analyses are conducted.

The system operates in a batch execution style: the user submits a job, which runs on a fixed set of files, and collects the output at the end of the run. The input format and location (typically a set of files in GFS) and the output destination are specified outside the language, as arguments to the command that submits the job to the system.

The command is called `saw` and its basic parameters define the name of the Sawzall program to be run, a pattern that matches the names of the files that hold the data records to be processed, and a set of files to receive the output. A typical job might be instantiated like this:

```
saw --program code.szl \
    --workqueue testing \
    --input_files /gfs/cluster1/2005-02-0[1-7]/submits.* \
    --destination /gfs/cluster2/$USER/output@100
```

The `program` flag names the file containing the Sawzall source of the program to be run, while the `workqueue` flag names the Workqueue cluster on which to run it. The `input_files` argument accepts standard Unix shell file-name-matching metacharacters to identify the files to be processed. The `destination` argument names the output files to be generated, using the notation that an at sign (`@`) introduces the number of files across which to distribute the results.

After the run, the user collects the data using a tool that accepts a `source` argument with the same notation as the `destination` argument to `saw`:

```
dump --source /gfs/cluster2/$USER/output@100 --format csv
```

This program merges the output data distributed across the named files and prints the final results in the specified format.

When a `saw` job request is received by the system, a Sawzall processor is invoked to verify that the program is syntactically valid. If it is, the source code is sent to the Workqueue machines for execution. The number of such machines is determined by looking at the size of the input and the number of input files, with a default upper limit proportional to the size of the Workqueue. A flag to `saw` can override this behavior and set the number explicitly.

These machines each compile the Sawzall program and run it on their portion of the input data, one record at a time. The emitted values are passed to the Sawzall runtime system, which collects them and performs an initial stage of aggregation locally, to reduce the amount of intermediate data sent to the aggregation machines. The runtime sends the data when the memory used for intermediate values reaches a threshold, or when input processing is complete.

The aggregation machines, whose number is determined by the count in the `destination` flag to `saw`, collect the intermediate outputs from the execution machines and merge them to each produce a sorted subset of the output. To ensure the merging process sees all the necessary data, the intermediate outputs are assigned to aggregation machines deterministically according to the particular table and any index values it may have. Each aggregation machine's output is written to a file in GFS; the output of the entire job is a set of files, one per aggregation machine. Aggregation is therefore a parallel operation that can take full advantage of the cluster's resources.

In general, the generation of the output is fairly evenly spread across the aggregation machines. Each element of a table has relatively modest size and the elements are spread across the different outputs. Collection tables are an exception; such tables can be of arbitrary size. To avoid overloading a single aggregation machine, the values inside a collection table are spread across the aggregation machines in a round-robin fashion. No aggregation machine sees all the intermediate values, but collection tables by definition do not combine duplicate results so this is not a problem.

Values stored by the aggregation machines are not in final form, but instead in an intermediate form carrying information to facilitate merging it with other values. In fact, this same intermediate form is used to pass the intermediate values from the execution machines to the aggregation machines. By leaving the values in an intermediate form, they can be merged with the values from another job. Large Sawzall jobs can be broken into pieces to be run separately, with final results constructed by merging the elements in a final phase. (This is one advantage of this system over plain MapReduce; even MapReduce can have problems with jobs that run for days or weeks of real time.)

This multi-stage merge—at execution machines, aggregations machines, and perhaps across jobs—is the reason the aggregators must be associative to work well. They must also be commutative, since the order in which the input records are processed is undefined, as is the order of merging of the intermediate values.

Typically, the data resulting from an analysis is much smaller than the input, but there are important examples where this is not the case. For instance, a program could use an indexed collection table to organize the input along some relevant axis, in which case the output would be as large as the input. Such transformations are handled efficiently even for large data sets. The input and the output are evenly partitioned across the execution and aggregation engines respectively. Each aggregation engine sorts its portion of the output data using a disk-based sort. The resulting data can then be combined efficiently to produce the final output using a merge sort.

8.1 Implementation

The Sawzall language is implemented as a conventional compiler, written in C++, whose target language is an interpreted instruction set, or byte-code. The compiler and the byte-code interpreter are part of the same binary, so the user presents source code to Sawzall and the system runs it directly. (This is a common but not universal design; Java for instance splits its compiler and interpreter into separate programs.) Actually, the Sawzall language system is structured as a library with an external interface that accepts source code and compiles and runs it, along with bindings to connect to externally-provided aggregators.

Those aggregators are implemented by `saw`, which is a program that links to the Sawzall compiler and interpreter library and is in turn implemented above MapReduce, running Sawzall in the map phase and the aggregators in the reduce phase. MapReduce manages the distribution of execution and aggregation machines, locates the computation near the GFS machines holding the data to minimize network traffic and thereby improve throughput, and handles machine failure and other faults. The system is therefore a "MapReduction" [8], but an unusual one. It actually incorporates two instances of MapReduce.

The first, quick job runs in parallel on the Workqueue to evaluate the size of the input and set up the main job. (There may be many thousands of input files and it's worth checking their size in parallel.) The second job then runs Sawzall. `Saw` thus acts in part as a convenient wrapper for MapReduce, setting up the job automatically based on the input rather than the usual method involving a number of user-specified flags. More important is the convenience of Sawzall and its aggregators. Leaving aside the comfort afforded by its simple programming model, Sawzall is implemented within a single binary that runs whatever code is presented to it; by contrast, other MapReductions must be written in C++ and compiled separately. Moreover, the various aggregators in Sawzall provide functionality that significantly extends the power of MapReduce. It would be possible to create a MapReduce wrapper library that contained the implementation of `top` tables and others, but even then they would not be nearly as simple or convenient to use as they are from Sawzall. Moreover, the ideas for some of those aggregators grew out of the system model provided by Sawzall. Although restrictive, the focus required by the Sawzall execution model can lead to unusual and creative programming ideas.

8.2 Chaining

One common use of the output of a Sawzall job is to inject the resulting data into a traditional relational database for subsequent analysis. This is usually done by a separate custom program, perhaps written in Python, that transforms the data into the SQL code to create the table. We might one day provide a more direct way to achieve this injection.

Sometimes the result of a Sawzall job is provided as input to another, by a process called *chaining*. A simple example is to calculate the exact "top 10" list for some input. The Sawzall `top` table is efficient but approximate. If the exact results are important, they can be derived by a two-step process. The first step creates an indexed `sum` table to count the frequency of input values; the second uses a `maximum` table to select the most popular. The steps are separate Sawzall jobs and the first step must run to completion before the second step can commence. Although a little clumsy, chaining is an effective way to extend the data analysis tasks that can be expressed in Sawzall.

9 More Examples

Here is another complete example that illustrates how Sawzall is used in practice. It processes a web document repository to answer the question: for each web domain, which page has the highest PageRank [14]? Roughly speaking, which is the most linked-to page?

```
proto "document.proto"

max_pagerank_url:
     table maximum(1) [domain: string] of url: string
           weight pagerank: int;

doc: Document = input;

emit max_pagerank_url[domain(doc.url)] <- doc.url
     weight doc.pagerank;
```

The protocol buffer format is defined in the file `"document.proto"`. The table is called `max_pagerank_url` and will record the highest-weighted value emitted for each index. The index is the domain, the value is the URL, and the weight is the document's Page Rank. The program parses the input record and then does a relatively sophisticated `emit` statement. It calls the library function `domain(doc.url)` to extract the domain of the URL to use as the index, emits the URL itself as the value, and uses the Page Rank for the document as the weight.

When this program is run over the repository, it shows the expected result that for most sites, the most-linked page is `www.`*site*`.com`—but there are surprises. The Acrobat download site is the top page for `adobe.com`, while those who link to `banknotes.com` go right to the image gallery and `bangkok-th.com` pulls right to the `Night_Life` page.

Because Sawzall makes it easy to express calculations like this, the program is nice and short. Even using MapReduce, the equivalent straight C++ program is about a hundred lines long.

Here is an example with a multiply-indexed aggregator. We wish to look at a set of search query logs and construct a map showing how the queries are distributed around the globe.

```
proto "querylog.proto"

queries_per_degree: table sum[lat: int][lon: int] of int;
```

```
    log_record: QueryLogProto = input;

    loc: Location = locationinfo(log_record.ip);
    emit queries_per_degree[int(loc.lat)][int(loc.lon)] <- 1;
```

The program is straightforward. We import the DDL for the query logs, declare a table indexed by integer latitude and longitude and extract the query from the log. Then we use a built-in function that looks up the incoming IP address in a database to recover the location of the requesting machine (probably its ISP), and then count `1` for the appropriate latitude/longitude bucket. The expression `int(loc.lat)` converts `loc.lat`, a `float`, to an integer, thereby truncating it to the degree and making it suitable as an index value. For a higher-resolution map, a more sophisticated calculation would be required.

The output of this program is an array of values suitable for creating a map, as in Figure 4.

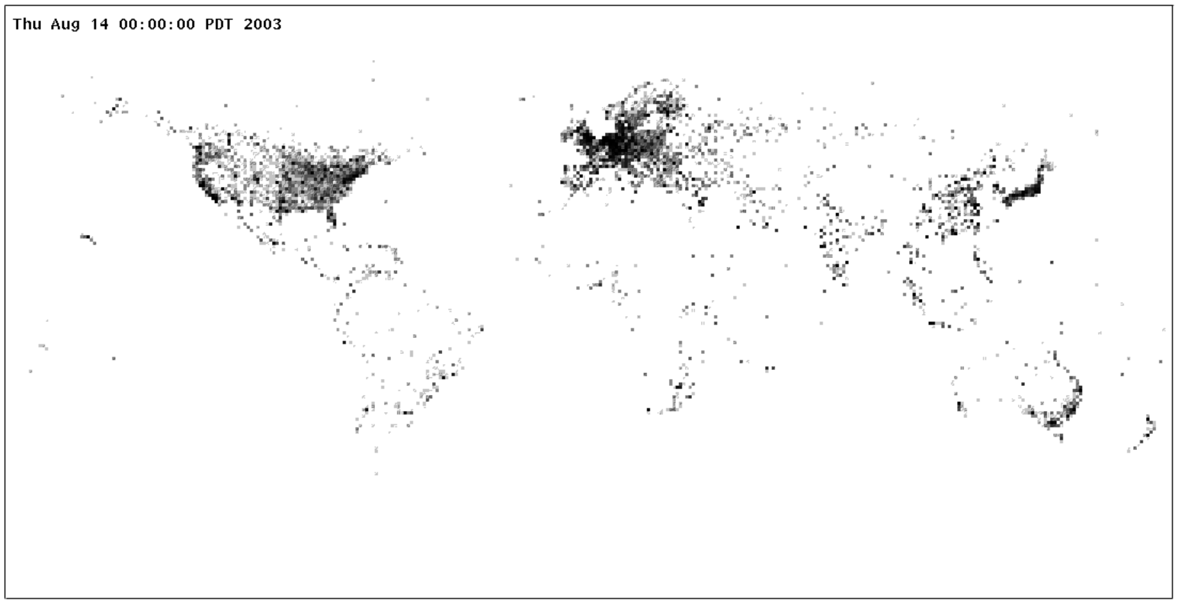


Figure 4: Query distribution.

10 Execution Model

At the statement level, Sawzall is an ordinary-looking language, but from a higher perspective it has several unusual properties, all serving the goal of enabling parallelism.

The most important, of course, is that it runs on one record at a time. This means it has no memory of other records it has processed (except through values emitted to the aggregators, outside the language). It is routine for a Sawzall job to be executing on a thousand machines simultaneously, yet the system requires no explicit communication between those machines. The only communication is from the Sawzall executions to the downstream aggregators.

To reinforce this point, consider the problem of counting input records. As we saw before, this program,

```
count: table sum of int;
emit count <- 1;
```

will achieve the job. By contrast, consider this erroneous program, which superficially looks like it should work:

```
count: int = 0;
count++;
```

This program fails to count the records because, *for each record,* it sets `count` to zero, increments it, and throws the result away. Running on many machines in parallel, it will throw away all the information with great efficiency.

The Sawzall program is reset to the initial state at the beginning of processing for each record. Correspondingly, after processing a record and emitting all relevant data, any resources consumed during execution—variables, temporaries, etc.—can be discarded. Sawzall therefore uses an arena allocator [11], resetting the arena to the initial state after each record is processed.

Sometimes it is useful to do non-trivial initialization before beginning execution. For instance, one might want to create a large array or map to be queried when analyzing each record. To avoid doing such initialization for every record, Sawzall has a declaration keyword `static` that asserts that the variable is to be initialized once (per runtime instance) and considered part of the initial runtime state for processing each record. Here is a trivial example:

```
static CJK: map[string] of string = {
    "zh": "Chinese",
    "ja": "Japanese",
    "ko": "Korean",
};
```

The `CJK` variable will be created during initialization and its value stored permanently as part of the initial state for processing each record.

The language has no reference types; it has pure value semantics. Even arrays and maps behave as values. (The implementation uses copy-on-write using reference counting to make this efficient in most cases.) Although in general this can lead to some awkwardness—to modify an array in a function, the function must return the array—for the typical Sawzall program the issue doesn't arise. In return, there is the possibility to parallelize the processing within a given record without the need for fine-grained synchronization or the fear of aliasing, an opportunity the implementation does not yet exploit.

11 Domain-specific Properties of the Language

Sawzall has a number of properties selected specifically for the problem domain in which it operates. Some have already been discussed; this section examines a few others.

First, atypical of most such “little languages” [2], Sawzall is statically typed. The main reason is dependability. Sawzall programs can consume hours, even months, of CPU time in a single run, and a late-arising dynamic type error can be expensive. There are other, less obvious reasons too. It helps the implementation of the aggregators to have their type fixed when they are created. A similar argument applies to the parsing of the input protocol buffers; it helps to know the expected input type precisely. Also, it is likely that overall performance of the interpreter is improved by avoiding dynamic type-checking at run time, although we have not attempted to verify this claim. Finally, compile-time type checking and the absence of automatic conversions require the programmer to be explicit about type conversions. The only exception is variable initialization, but in that case the type is still explicit and the program remains type-safe.

Static typing guarantees that the types of variables are always known, while permitting convenience at initialization time. It is helpful that initializations like

```
    t: time = "Apr 1 12:00:00 PST 2005";
```

are easy to understand, yet also type-safe.

The set of basic types and their properties are also somewhat domain-dependent. The presence of time as a basic type is motivated by the handling of time stamps from log records; it is luxurious to have a language that handles Daylight Savings Time correctly. More important (but less unusual these days), the language defines a Unicode representation for strings, yet handles a variety of external character encodings.

Two novel features of the language grew out of the demands of analyzing large data sets: the handling of undefined values, and logical quantifiers. The next two sections describe these in detail.

11.1 Undefined values

Sawzall does not provide any form of exception processing. Instead, it has the notion of *undefined values*, a representation of erroneous or indeterminate results including division by zero, conversion errors, I/O problems, and so on. If the program attempts to read an undefined value outside of an initialization, it will crash and provide a report of the failure.

A predicate, `def()`, can be used to test if a value is defined; it returns `true` for a defined value and `false` for an undefined value. The idiom for its use is the following:

```
v: Value = maybe_undefined();
if (def(v)) {
    calculation_using(v);
}
```

Here's an example that must handle undefined values. Let's extend the query-mapping program by adding a time axis to the data. The original program used the function `locationinfo()` to discover the location of an IP address using an external database. That program was unsafe because it would fail if the IP address could not be found in the database. In that case, `locationinfo()` would return an undefined value, but we can protect against that using a `def()` predicate.

Here is the expanded, robust, program:

```
proto "querylog.proto"

static RESOLUTION: int = 5;  # minutes; must be divisor of 60

log_record: QueryLogProto = input;

queries_per_degree: table sum[t: time][lat: int][lon: int] of int;

loc: Location = locationinfo(log_record.ip);
if (def(loc)) {
    t: time = log_record.time_usec;
    m: int = minuteof(t);  # within the hour
    m = m - m % RESOLUTION;
    t = trunctohour(t) + time(m * int(MINUTE));
    emit queries_per_degree[t][int(loc.lat)][int(loc.lon)] <- 1;
}
```

(Notice that we just disregard the record if it has no known location; this seems a fine way to handle it.) The calculation in the `if` statement uses some built-in functions (and the built-in constant `MINUTE`) to truncate the microsecond-resolution time stamp of the record to the boundary of a 5-minute time bucket.

Given query log records spanning an extended period, this program will extract data suitable for constructing a movie showing how activity changes over time.[2]

A careful programmer will use `def()` to guard against the usual errors that can arise, but sometimes the error syndrome can be so bizarre that a programmer would never think of it. If a program is processing a terabyte of data, there may be some records that contain surprises; often the quality of the data set may be outside the control of the person doing the analysis, or contain occasional

[2]The movie can be seen at `http://labs.google.com/papers/sawzall.html`.

unusual values beyond the scope of the current analysis. Particularly for the sort of processing for which Sawzall is used, it will often be perfectly safe just to disregard such erroneous values.

Sawzall therefore provides a mode, set by a run-time flag, that changes the default behavior of undefined values. Normally, using an undefined value (other than in an initialization or `def()` test) will terminate the program with an error report. When the run-time flag is set, however, Sawzall simply elides all statements that depend on the undefined value. To be precise, if a statement's computation references the undefined value in any way, the statement is skipped and execution is resumed after that statement. Simple statements are just dropped. An `if` statement with an undefined condition will be dropped completely. If a loop's condition becomes undefined while the loop is active, the loop is terminated but the results of previous iterations are unaffected.

For the corrupted record, it's as though the elements of the calculation that depended on the bad value were temporarily removed from the program. Any time such an elision occurs, the run-time will record the circumstances in a special pre-defined `collection` table that serves as a log of these errors. When the run completes the user may check whether the error rate was low enough to accept the results that remain.

One way to use the flag is to debug with it off—otherwise bugs will vanish—but when it comes time to run in production on a large data set, turn the flag on so that the program won't be broken by crazy data.

This is an unusual way to treat errors, but it is very convenient in practice. The idea is related to some independent work by Rinard *et al.* [15] in which the `gcc` C compiler was modified to generate code that protected against certain errors. For example, if a program indexes off the end of an array, the generated code will make up values and the program can continue obliviously. This peculiar behavior has the empirical property of making programs like web servers much more robust against failure, even in the face of malicious attacks. The handling of undefined values in Sawzall adds a similar level of robustness.

11.2 Quantifiers

Although Sawzall operates on individual records, those records sometimes contain arrays or other structures that must be analyzed as a unit to discover some property. Is there an array element with this value? Do all the values satisfy some condition? To make these ideas easy to express, Sawzall provides *logical quantifiers*, a special notation analogous to the "for each", "for any", and "for all" quantifiers of mathematics.

Within a special construct, called a `when` statement, one defines a quantifier, a variable, and a boolean condition using the variable. If the condition is satisfied, the associated statement is executed.

Quantifier variables are declared like regular variables, but the base type (usually `int`) is prefixed by a keyword specifying the form of quantifier. For example, given an array `a`, the statement

```
when (i: some int; B(a[i]))
    F(i);
```

executes `F(i)` if and only if, for *some* value of `i`, the boolean condition `B(a[i])` is true. When `F(i)` is invoked, `i` will be bound to the value that satisfies the condition.

There are three quantifier types: `some`, which executes the statement if the condition is true for any value (if more than one satisfies, an arbitrary choice is made); `each`, which executes the statement for all the values that satisfy the condition; and `all`, which executes the statement if the condition is true for all valid values of the quantifier (and does not bind the variable in the statement).

Sawzall analyzes the conditional expression to discover how to limit the range of evaluation of a `when` statement. It is a compile-time error if the condition does not provide enough information to construct a reasonable evaluator for the quantifier. In the example above, the quantifier variable `i` is used as an index in an array expression `a[i]`, which is sufficient to define the range. For more complex conditions, the system uses the `def()` operator internally to explore the bounds safely if necessary. (Indexing beyond the bounds of an array results in an undefined value.) Consider for example scanning a sparse multidimensional array `a[i][j]`. In such an expression, the two quantifier variables `i` and `j` must pairwise define a valid entry and the implementation can use `def` to explore the matrix safely.

`When` statements may contain multiple quantifier variables, which in general can introduce ambiguities of logic programming [6]. Sawzall defines that if multiple variables exist, they will be bound and evaluated in the order declared. This ordering, combined with the restricted set of quantifiers, is sufficient to eliminate the ambiguities.

Here are a couple of examples. The first tests whether two arrays share a common element:

```
when(i, j: some int; a1[i] == a2[j]) {
        ...
}
```

The second expands on this. Using array slicing, indicated by `:` notation in array indices, it tests whether two arrays share a common subarray of three or more elements:

```
when(i0, i1, j0, j1: some int; a[i0:i1] == b[j0:j1] &&
                               i1 >= i0+3) {
    ...
}
```

It is convenient not to write all the code to handle the edge conditions in such a test. Even if the arrays are shorter than three elements, this statement will function correctly; the evaluation of `when` statements guarantees safe execution.

In principle, the evaluation of a `when` statement is parallelizable, but we have not explored this option yet.

12 Performance

Although Sawzall is interpreted, that is rarely the limiting factor in its performance. Most Sawzall jobs do very little processing per record and are therefore I/O bound; for most of the rest, the CPU spends the majority of its time in various run-time operations such as parsing protocol buffers.

Nevertheless, to compare the single-CPU speed of the Sawzall interpreter with that of other interpreted languages, we wrote a couple of microbenchmarks. The first computes pixel values for displaying the Mandelbrot set, to measure basic arithmetic and loop performance. The second measures function invocation using a recursive function to calculate the first 35 Fibonacci numbers. We ran the tests on a 2.8GHz x86 desktop machine. The results, shown in Table 1, demonstrate Sawzall is significantly faster than Python, Ruby, or Perl, at least for these microbenchmarks. (We are somewhat surprised by this; the other languages are older and we would expect them to have finely tuned implementations.) On the other hand, for these microbenchmarks Sawzall is about 1.6 times slower than interpreted Java, 21 times slower than compiled Java, and 51 times slower than compiled C++.[3]

	Sawzall	Python	Ruby	Perl
Mandelbrot run time	12.09s	45.42s	73.59s	38.68s
factor	1.00	3.75	6.09	3.20
Fibonacci run time	11.73s	38.12s	47.18s	75.73s
factor	1.00	3.24	4.02	6.46

Table 1: Microbenchmarks. The first is a Mandelbrot set calculation: 500×500 pixel image with 500 iterations per pixel maximum. The second uses a recursive function to calculate the first 35 Fibonacci numbers.

The key performance metric for the system is not the single-machine speed but how the speed scales as we add machines when processing a large data set. We took a 450GB sample of compressed query log data and ran a Sawzall program over it to count the occurrences of certain words. The core of the program looked schematically like this:

```
result: table sum[key: string][month: int][day: int] of int;
static keywords: array of string =
  { "hitchhiker", "benedict", "vytorin",
    "itanium", "aardvark" };
```

[3]The Java implementation was Sun's client Java™1.3.1_02 with HotSpot™; C++ was `gcc` 3.2.2

```
querywords: array of string = words_from_query();
month: int = month_of_query();
day: int = day_of_query();

when (i: each int; j: some int; querywords[i] == keywords[j])
    emit result[keywords[j]][month][day] <- 1;
```

We ran the test on sets of machines varying from 50 2.4GHz Xeon computers to 600. The timing numbers are graphed in Figure 5. At 600 machines, the aggregate throughput was 1.06GB/s of compressed data or about 3.2GB/s of raw input. If scaling were perfect, performance would be proportional to the number of machines, that is, adding one machine would contribute one machine's worth of throughput. In our test, the effect is to contribute 0.98 machines.

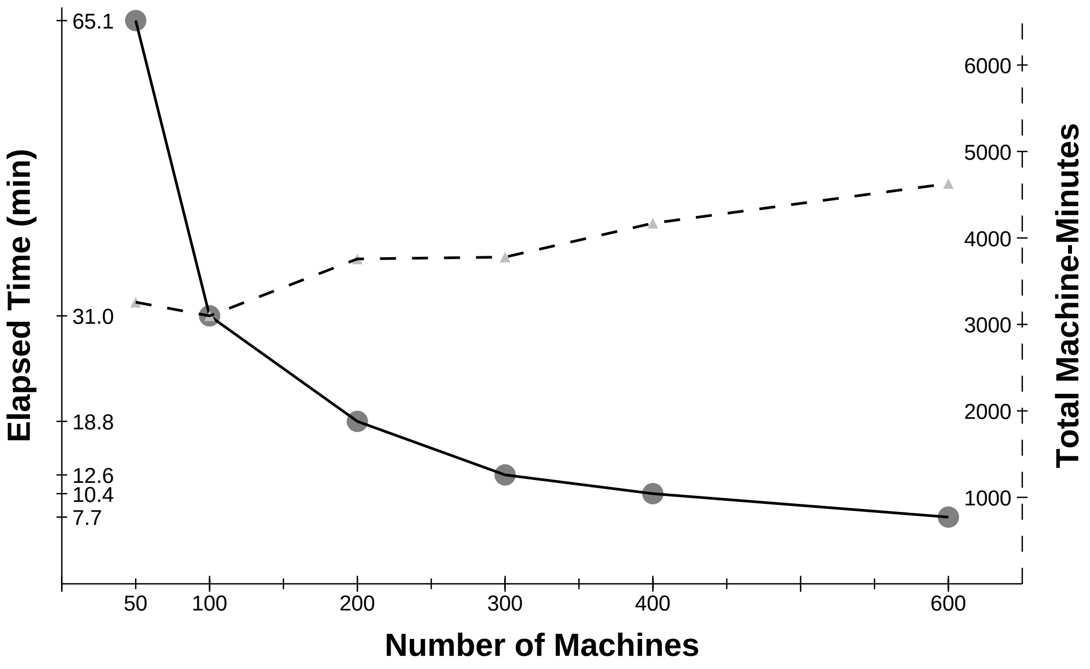


Figure 5: Performance scales well as we add machines. The solid line is elapsed time; the dashed line is the product of machines and elapsed time. Over the range of 50 to 600 machines, the machine-minutes product degrades only 30%.

13 Why a new language?

Why put a language above MapReduce? MapReduce is very effective; what's missing? And why a *new* language? Why not just attach an existing language such as Python to MapReduce?

The usual reasons for creating a special-purpose language apply here. Notation customized to a particular problem domain can make programs clearer, more compact, and more expressive. Capturing the aggregators in the language (and its environment) means that the programmer never has to provide one, unlike when using MapReduce. It also has led to some elegant programs as well as a comfortable way to think about data processing problems in large distributed data sets. Also, support for protocol buffers and other domain-specific types simplifies programming at a lower level. Overall, Sawzall programs tend to be around 10 to 20 times shorter than the equivalent MapReduce programs in C++ and significantly easier to write.

Other advantages of a custom language include the ability to add domain-specific features, custom debugging and profiling interfaces, and so on.

The original motivation, however, was completely different: parallelism. Separating out the aggregators and providing no other inter-record analysis maximizes the opportunity to distribute processing across records. It also provides a model for distributed processing, which in turn encourages users to think about the problem in a different light. In an existing language such as Awk [12] or Python [1], users would be tempted to write the aggregators in that language, which would be difficult to parallelize. Even if one provided a clean interface and library for aggregators in these languages, seasoned users would want to roll their own sometimes, which could introduce dramatic performance problems.

The model that Sawzall provides has proven valuable. Although some problems, such as database joins, are poor fits to the model, most of the processing we do on large data sets fits well and the benefit in notation, convenience, and expressiveness has made Sawzall a popular language at Google.

One unexpected benefit of the system arose from the constraints the programming model places on the user. Since the data flow from the input records to the Sawzall program is so well structured, it was easy to adapt it to provide fine-grained access control to individual fields within records. The system can automatically and securely wrap the user's program with a layer, itself implemented in Sawzall, that elides any sensitive fields. For instance, production engineers can be granted access to performance and monitoring information without exposing any traffic data. The details are outside the scope of this paper.

14 Utility

Although it has been deployed for only about 18 months, Sawzall has become one of the most widely used programming languages at Google. There are over two thousand Sawzall source files registered in our source code control system and a growing collection of libraries to aid the processing of various special-purpose data sets. Most Sawzall programs are of modest size, but the largest are several thousand lines long and generate dozens of multi-dimensional tables in a single run.

One measure of Sawzall's utility is how much data processing it does. We monitored its use during the month of March 2005. During that time, on one dedicated Workqueue cluster with 1500 Xeon CPUs, there were 32,580 Sawzall jobs launched, using an average of 220 machines each. While running those jobs, 18,636 failures occurred (application failure, network outage, system crash, etc.) that triggered rerunning some portion of the job. The jobs read a total of 3.2×10^{15} bytes of data (2.8PB) and wrote 9.9×10^{12} bytes (9.3TB) (demonstrating that the term "data reduction" has some resonance). The average job therefore processed about 100GB. The jobs collectively consumed almost exactly one machine-century.

15 Related Work

Traditional data processing is done by storing the information in a relational database and processing it with SQL queries. Our system has many differences. First, the data sets are usually too large to fit in a relational database; files are processed *in situ* rather than being imported into a database server. Also, there are no pre-computed tables or indices; instead the purpose of the system is to construct *ad hoc* tables and indices appropriate to the computation. Although the aggregators are fixed, the presence of a procedural language to process the data enables complex, sophisticated analyses.

Sawzall is very different from SQL, combining a fairly traditional procedural language with an interface to efficient aggregators that process the results of the analysis applied to each record. SQL is excellent at database join operations, while Sawzall is not. On the other hand, Sawzall can be run on a thousand or more machines in parallel to process enormous data sets.

Brook [3] is another language for data processing, specifically graphical image processing. Although different in application area, like Sawzall it enables parallelism through a one-element-at-a-time computation model and associative reduction kernels.

A different approach to the processing of large data stores is to analyze them with a data stream model. Such systems process the data as it flows in, and their operators are dependent on the order of the input records. For example, Aurora [4] is a stream processing system that supports a (potentially large) set of standing queries on streams of data. Analogous to Sawzall's predefinition of its aggregators, Aurora provides a small, fixed set of operators, although two of them are escapes to user-defined functions. These operators can be composed to create more interesting queries. Unlike Sawzall, some Aurora operators work on a contiguous sequence, or window, of input values. Aurora only keeps a limited amount of data on hand, and is not designed for querying large archival stores. There is a facility to add new queries to the system, but they only operate on the recent past. Aurora's efficiency comes from a carefully designed run-time system and a query optimizer, rather than Sawzall's brute force parallel style.

Another stream processing system, Hancock [7], goes further and provides extensive support for storing per-query intermediate state. This is quite a contrast to Sawzall, which deliberately reverts to its initialization state after each input record. Like Aurora, Hancock concentrates on efficient operation of a single thread instead of massive parallelism.

16 Future Work

The throughput of the system is very high when used as intended, with thousands of machines in parallel. Because the language is modest in scope, the programs written in it tend to be small and are usually I/O bound. Therefore, although it is an interpreted language, the implementation is fast enough most of the time. Still, some of the larger or more complex analyses would be helped by more aggressive compilation, perhaps to native machine code. The compiler would run once per machine and then apply the accelerated binary to each input record. Exploratory work on such a compiler is now underway.

Occasionally a program needs to query external databases when processing a record. While support is already provided for a common set of small databases, such as the IP location information, our system could profit from an interface to query an external database. Since Sawzall processes each record independently, the system could suspend processing of one record when the program makes such a query, and continue when the query is completed. There are obvious opportunities for parallelism in this design.

Our analyses sometimes require multiple passes over the data, either multiple Sawzall passes or Sawzall passes followed by processing by another system, such as a traditional database or a program written in a general-purpose programming language. Since the language does not directly support such "chaining", these multi-pass programs can be awkward to express in Sawzall. Language extensions are being developed that will make it easy to express the chaining of multiple passes, as are aggregator extensions allowing output to be delivered directly to external systems.

Some analyses join data from multiple input sources, often after a Sawzall preprocessing step or two. Joining is supported, but in general requires extra chaining steps. More direct support of join operations would simplify these programs.

A more radical system model would eliminate the batch-processing mode entirely. It would be convenient for tasks such as performance monitoring to have the input be fed continuously to the Sawzall program, with the aggregators keeping up with the data flow. The aggregators would be maintained in some on-line server that could be queried at any time to discover the current value of any table or table entry. This model is similar to the streaming database work [4][7] and was in fact the original thinking behind the system. However, before much implementation had been completed, the MapReduce library was created by Dean and Ghemawat and it proved so effective that the continuous system was never built. We hope to return to it one day.

17 Conclusions

When the problem is large enough, a new approach may be necessary. To make effective use of large computing clusters in the analysis of large data sets, it is helpful to restrict the programming model to guarantee high parallelism. The trick is to do so without unduly limiting the expressiveness of the model.

Our approach includes a new programming language called Sawzall. The language helps capture the programming model by forcing the programmer to think one record at a time, while providing an expressive interface to a novel set of aggregators that capture many common data processing and data reduction problems. In return for learning a new language, the user is rewarded by the ability to write short, clear programs that are guaranteed to work well on thousands of machines in parallel. Ironically—but vitally—the user needs to know nothing about parallel programming; the language and the underlying system take care of all the details.

It may seem paradoxical to use an interpreted language in a high-throughput environment, but we have found that the CPU time is rarely the limiting factor; the expressibility of the language means that most programs are small and spend most of their time in I/O and native run-time code. Moreover, the flexibility of an interpreted implementation has been helpful, both in ease of experimentation at the linguistic level and in allowing us to explore ways to distribute the calculation across many machines.

Perhaps the ultimate test of our system is scalability. We find linear growth in performance as we add machines, with a slope near unity. Big data sets need lots of machines; it's gratifying that lots of machines can translate into big throughput.

18 Acknowledgements

Geeta Chaudhry wrote the first significant Sawzall programs and gave invaluable feedback. Amit Patel, Paul Haahr and Greg Rae were enthusiastic and helpful early users. Paul Haahr created the

Page Rank example. Dick Sites and Renée French helped with the figures. We received helpful comments about the paper from Dan Bentley, Dave Hanson, Patrick Jordan, John Lamping, Dick Sites, Tom Szymanski, Deborah A. Wallach, and Lawrence You.

References

[1] David M. Beazley, Python Essential Reference, New Riders, Indianapolis, 2000.

[2] Jon Bentley, Programming Pearls, CACM August 1986 v 29 n 8 pp. 711-721.

[3] Ian Buck et al., Brook for GPUs: Stream Computing on Graphics Hardware, Proc. SIGGRAPH, Los Angeles, 2004.

[4] Don Carney et al., Monitoring Streams – A New Class of Data Management Applications, Brown Computer Science Technical Report TR-CS-02-04. At `http://www.cs.brown.edu/research/aurora/aurora_tr.pdf`.

[5] M. Charikar, K. Chen, and M. Farach-Colton, Finding frequent items in data streams, Proc 29th Intl. Colloq. on Automata, Languages and Programming, 2002.

[6] W. F. Clocksin and C. S. Mellish, Programming in Prolog, Springer, 1994.

[7] Cortes et al., Hancock: A Language for Extracting Signatures from Data Streams, Proc. Sixth International Conference on Knowledge Discovery and Data Mining, Boston, 2000, pp. 9-17.

[8] Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, Proc 6th Symposium on Operating Systems Design and Implementation, San Francisco, 2004, pages 137-149.

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, The Google File System, Proc. 19th Symposium on Operating System Principles, Lake George, New York, 2003, pp. 29-43.

[10] M. Greenwald and S. Khanna, Space-efficient online computation of quantile summaries, Proc. SIGMOD, Santa Barbara, CA, May 2001, pp. 58-66.

[11] David R. Hanson, Fast allocation and deallocation of memory based on object lifetimes. Software–Practice and Experience, 20(1):512, January 1990.

[12] Brian Kernighan, Peter Weinberger, and Alfred Aho, The AWK Programming Language, Addison-Wesley, Massachusetts, 1988.

[13] Brian Kernighan, personal communication.

[14] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd, The pagerank citation algorithm: bringing order to the web, Proc. of the Seventh conference on the World Wide Web, Brisbane, Australia, April 1998.

[15] Martin Rinard et al., Enhancing Server Reliability Through Failure-Oblivious Computing, Proc. Sixth Symposium on Operating Systems Design and Implementation, San Francisco, 2004, pp. 303-316.

[16] Douglas Thain, Todd Tannenbaum, and Miron Livny, Distributed computing in practice: The Condor experience, Concurrency and Computation: Practice and Experience, 2004.

The SMAQ stack for big data

Storage, MapReduce and Query are ushering in data-driven products and services.

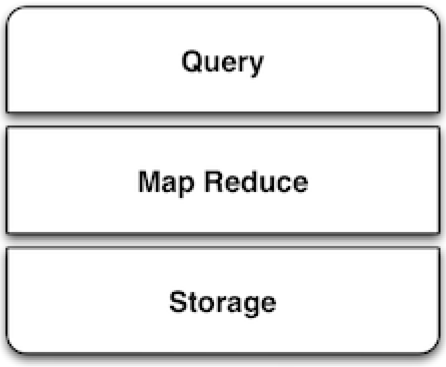
by Edd Dumbill | @edd | Comments: 16 | 22 September 2010

SMAQ report sections

"Big data" is data that becomes large enough that it cannot be processed using conventional methods. Creators of web search engines were among the first to confront this problem. Today, social networks, mobile phones, sensors and science contribute to petabytes of data created daily.

To meet the challenge of processing such large data sets, Google created MapReduce. Google's work and Yahoo's creation of the Hadoop MapReduce implementation has spawned an ecosystem of big data processing tools.

As MapReduce has grown in popularity, a stack for big data systems has emerged, comprising layers of Storage, MapReduce and Query (SMAQ). SMAQ systems are typically open source, distributed, and run on commodity hardware.

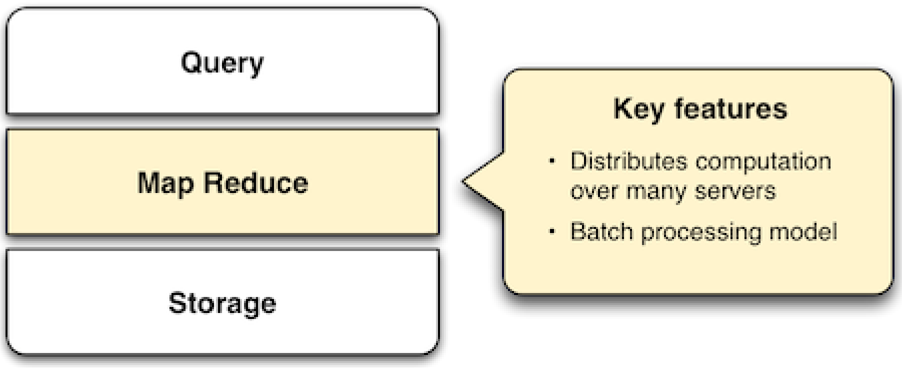


In the same way the commodity LAMP stack of Linux, Apache, MySQL and PHP changed the landscape of web applications, SMAQ systems are bringing commodity big data processing to a broad audience. SMAQ systems underpin a new era of innovative data-driven products and services, in the same way that LAMP was a critical enabler for Web 2.0.

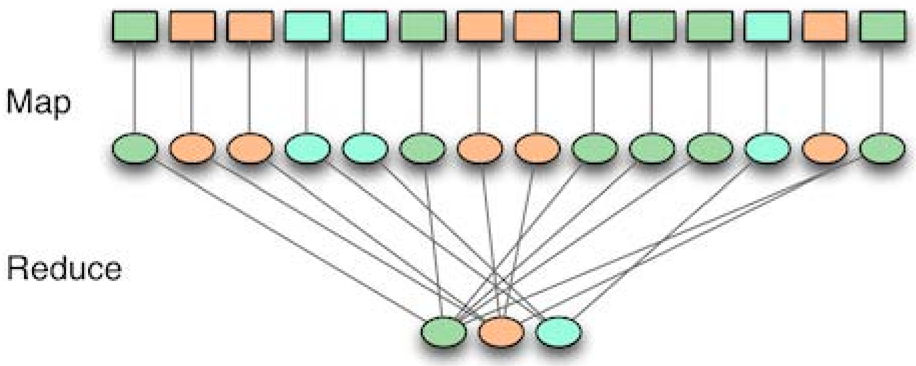
Though dominated by Hadoop-based architectures, SMAQ encompasses a variety of systems, including leading NoSQL databases. This paper describes the SMAQ stack and where today's big data tools fit into the picture.

MapReduce

Created at Google in response to the problem of creating web search indexes, the MapReduce framework is the powerhouse behind most of today's big data processing. The key innovation of MapReduce is the ability to take a query over a data set, divide it, and run it in parallel over many nodes. This distribution solves the issue of data too large to fit onto a single machine.



To understand how MapReduce works, look at the two phases suggested by its name. In the map phase, input data is processed, item by item, and transformed into an intermediate data set. In the reduce phase, these intermediate results are reduced to a summarized data set, which is the desired end result.



A simple example of MapReduce is the task of counting the number of unique words in a document. In the map phase, each word is identified and given the count of 1. In the reduce phase, the counts are added together for each word.

If that seems like an obscure way of doing a simple task, that's because it is. In order for MapReduce to do its job, the map and reduce phases must obey certain constraints that allow the work to be parallelized. Translating queries into one or more MapReduce steps is not an intuitive process. Higher-level abstractions have been developed to ease this, discussed under Query below.

An important way in which MapReduce-based systems differ from conventional databases is that they process data in a batch-oriented fashion. Work must be queued for execution, and may take minutes or hours to process.

Using MapReduce to solve problems entails three distinct operations:

- **Loading the data** -- This operation is more properly called Extract, Transform, Load (ETL) in data warehousing terminology. Data must be extracted from its source, structured to make it ready for processing, and loaded into the storage layer for MapReduce to operate on it.
- **MapReduce** -- This phase will retrieve data from storage, process it, and return the results to the storage.
- **Extracting the result** -- Once processing is complete, for the result to be useful to humans, it must be retrieved from the storage and presented.

Many SMAQ systems have features designed to simplify the operation of each of these stages.

Hadoop MapReduce

Hadoop is the dominant open source MapReduce implementation. Funded by Yahoo, it emerged in 2006 and, according to its creator Doug Cutting, reached “web scale” capability in early 2008.

The Hadoop project is now hosted by Apache. It has grown into a large endeavor, with multiple subprojects that together comprise a full SMAQ stack.

Since it is implemented in Java, Hadoop's MapReduce implementation is accessible from the Java programming language. Creating MapReduce jobs involves writing functions to encapsulate the map and reduce stages of the computation. The data to be processed must be loaded into the Hadoop Distributed Filesystem.

Taking the word-count example from above, a suitable map function might look like the following (taken from the Hadoop MapReduce documentation, the key operations shown in bold).

```
public static class Map

        extends Mapper<LongWritable, Text, Text, IntWritable> {

        private final static IntWritable one = new IntWritable(1);

        private Text word = new Text();

        public void map(LongWritable key, Text value, Context context)

            throws IOException, InterruptedException {

                String line = value.toString();

                StringTokenizer tokenizer = new StringTokenizer(line);

                while (tokenizer.hasMoreTokens()) {

                        word.set(tokenizer.nextToken());

                        context.write(word, one);

                }

        }

}
```

The corresponding reduce function sums the counts for each word.

```
public static class Reduce

                extends Reducer<Text, IntWritable, Text, IntWritable> {

        public void reduce(Text key, Iterable<IntWritable> values,

                Context context) throws IOException, InterruptedException {

                int sum = 0;

                for (IntWritable val : values) {

                        sum += val.get();

                }

                context.write(key, new IntWritable(sum));

        }

}
```

The process of running a MapReduce job with Hadoop involves the following steps:

- Defining the MapReduce stages in a Java program
- Loading the data into the filesystem
- Submitting the job for execution
- Retrieving the results from the filesystem

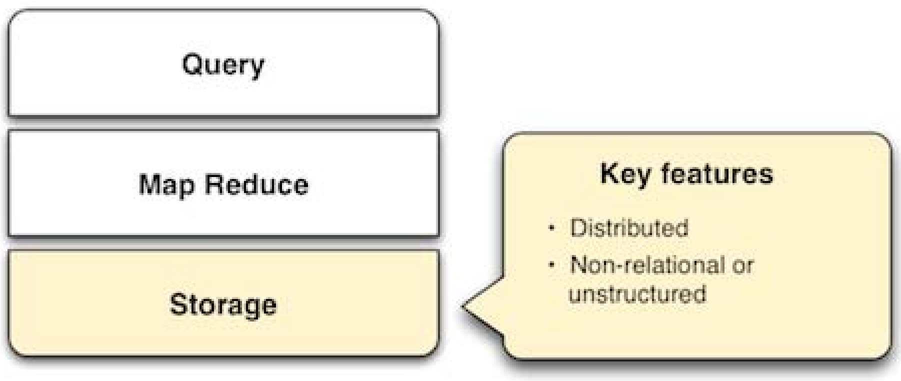
Run via the standalone Java API, Hadoop MapReduce jobs can be complex to create, and necessitate programmer involvement. A broad ecosystem has grown up around Hadoop to make the task of loading and processing data more straightforward.

Other implementations

MapReduce has been implemented in a variety of other programming languages and systems, a list of which may be found in Wikipedia's entry for MapReduce. Notably, several NoSQL database systems have integrated MapReduce, and are described later in this paper.

Storage

MapReduce requires storage from which to fetch data and in which to store the results of the computation. The data expected by MapReduce is not relational data, as used by conventional databases. Instead, data is consumed in chunks, which are then divided among nodes and fed to the map phase as key-value pairs. This data does not require a schema, and may be unstructured. However, the data must be available in a distributed fashion, to serve each processing node.



The design and features of the storage layer are important not just because of the interface with MapReduce, but also because they affect the ease with which data can be loaded and the results of computation extracted and searched.

Hadoop Distributed File System

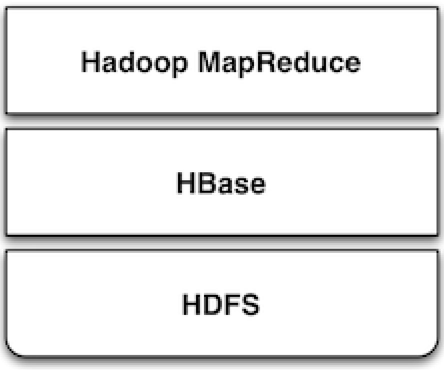
The standard storage mechanism used by Hadoop is the Hadoop Distributed File System, HDFS. A core part of Hadoop, HDFS has the following features, as detailed in the HDFS design document.

- **Fault tolerance** -- Assuming that failure will happen allows HDFS to run on commodity hardware.
- **Streaming data access** -- HDFS is written with batch processing in mind, and emphasizes high throughput rather than random access to data.
- **Extreme scalability** -- HDFS will scale to petabytes; such an installation is in production use at Facebook.
- **Portability** -- HDFS is portable across operating systems.
- **Write once** -- By assuming a file will remain unchanged after it is written, HDFS simplifies replication and speeds up data throughput.
- **Locality of computation** -- Due to data volume, it is often much faster to move the program near to the data, and HDFS has features to facilitate this.

HDFS provides an interface similar to that of regular filesystems. Unlike a database, HDFS can only store and retrieve data, not index it. Simple random access to data is not possible. However, higher-level layers have been created to provide finer-grained functionality to Hadoop deployments, such as HBase.

HBase, the Hadoop Database

One approach to making HDFS more usable is HBase. Modeled after Google's BigTable database, HBase is a column-oriented database designed to store massive amounts of data. It belongs to the NoSQL universe of databases, and is similar to Cassandra and Hypertable.



HBase uses HDFS as a storage system, and thus is capable of storing a large volume of data through fault-tolerant, distributed nodes. Like similar column-store databases, HBase provides REST and Thrift based API access.

Because it creates indexes, HBase offers fast, random access to its contents, though with simple queries. For complex operations, HBase acts as both a *source* and a *sink* (destination for computed data) for Hadoop MapReduce. HBase thus allows systems to interface with Hadoop as a database, rather than the lower level of HDFS.

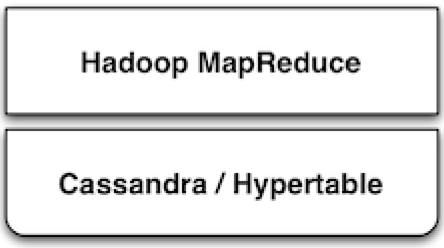
Hive

Data warehousing, or storing data in such a way as to make reporting and analysis easier, is an important application area for SMAQ systems. Developed originally at Facebook, Hive is a data warehouse framework built on top of Hadoop. Similar to HBase, Hive provides a table-based abstraction over HDFS and makes it easy to load structured data. In contrast to HBase, Hive can only run MapReduce jobs and is suited for batch data analysis. Hive provides a SQL-like query language to execute MapReduce jobs, described in the Query section below.

Cassandra and Hypertable

Cassandra and Hypertable are both scalable column-store databases that follow the pattern of BigTable, similar to HBase.

An Apache project, Cassandra originated at Facebook and is now in production in many large-scale websites, including Twitter, Facebook, Reddit and Digg. Hypertable was created at Zvents and spun out as an open source project.



Both databases offer interfaces to the Hadoop API that allow them to act as a source and a sink for MapReduce. At a higher level, Cassandra offers integration with the Pig query language (see the Query section below), and Hypertable has been integrated with Hive.

NoSQL database implementations of MapReduce

The storage solutions examined so far have all depended on Hadoop for MapReduce. Other NoSQL databases have built-in MapReduce features that allow computation to be parallelized over their data stores. In contrast with the multi-component SMAQ architectures of Hadoop-based systems, they offer a self-contained system comprising storage, MapReduce and query all in one.

Whereas Hadoop-based systems are most often used for batch-oriented analytical purposes, the usual function of NoSQL stores is to back live applications. The MapReduce functionality in these databases tends to be a secondary feature, augmenting other primary query mechanisms. Riak, for example, has a default timeout of 60 seconds on a MapReduce job, in contrast to the expectation of Hadoop that such a process may run for minutes or hours.

These prominent NoSQL databases contain MapReduce functionality:

- CouchDB is a distributed database, offering semi-structured document-based storage. Its key features include strong replication support and the ability to make distributed updates. Queries in CouchDB are implemented using JavaScript to define the map and reduce phases of a MapReduce process.
- MongoDB is very similar to CouchDB in nature, but with a stronger emphasis on performance, and less suitability for distributed updates, replication, and versioning. MongoDB MapReduce operations are specified using JavaScript.

- Riak is another database similar to CouchDB and MongoDB, but places its emphasis on high availability. MapReduce operations in Riak may be specified with JavaScript or Erlang.

Integration with SQL databases

In many applications, the primary source of data is in a relational database using platforms such as MySQL or Oracle. MapReduce is typically used with this data in two ways:

- Using relational data as a source (for example, a list of your friends in a social network).
- Re-injecting the results of a MapReduce operation into the database (for example, a list of product recommendations based on friends' interests).

It is therefore important to understand how MapReduce can interface with relational database systems. At the most basic level, delimited text files serve as an import and export format between relational databases and Hadoop systems, using a combination of SQL export commands and HDFS operations. More sophisticated tools do, however, exist.

The Sqoop tool is designed to import data from relational databases into Hadoop. It was developed by Cloudera, an enterprise-focused distributor of Hadoop platforms. Sqoop is database-agnostic, as it uses the Java JDBC database API. Tables can be imported either wholesale, or using queries to restrict the data import.

Sqoop also offers the ability to re-inject the results of MapReduce from HDFS back into a relational database. As HDFS is a filesystem, Sqoop expects delimited text files and transforms them into the SQL commands required to insert data into the database.

For Hadoop systems that utilize the Cascading API (see the Query section below) the cascading.jdbc and cascading-dbmigrate tools offer similar source and sink functionality.

Integration with streaming data sources

In addition to relational data sources, streaming data sources, such as web server log files or sensor output, constitute the most common source of input to big data systems. The Cloudera Flume project aims at providing convenient integration between Hadoop and streaming data sources. Flume aggregates data from both network and file sources, spread over a cluster of machines, and continuously pipes these into HDFS. The Scribe server, developed at Facebook, also offers similar functionality.

Commercial SMAQ solutions

Several massively parallel processing (MPP) database products have MapReduce functionality built in. MPP databases have a distributed architecture with independent nodes that run in parallel. Their primary application is in data warehousing and analytics, and they are commonly accessed using SQL.

- The Greenplum database is based on the open source PostreSQL DBMS, and runs on clusters of distributed hardware. The addition of MapReduce to the regular SQL interface enables fast, large-scale analytics over Greenplum databases, reducing query times by several orders of magnitude. Greenplum MapReduce permits the mixing of external data sources with the database storage. MapReduce operations can be expressed as functions in Perl or Python.
- Aster Data's nCluster data warehouse system also offers MapReduce functionality. MapReduce operations are invoked using Aster Data's SQL-MapReduce technology. SQL-MapReduce enables the intermingling of SQL queries with MapReduce jobs defined using code, which may be written in languages including C#, C++, Java, R or Python.

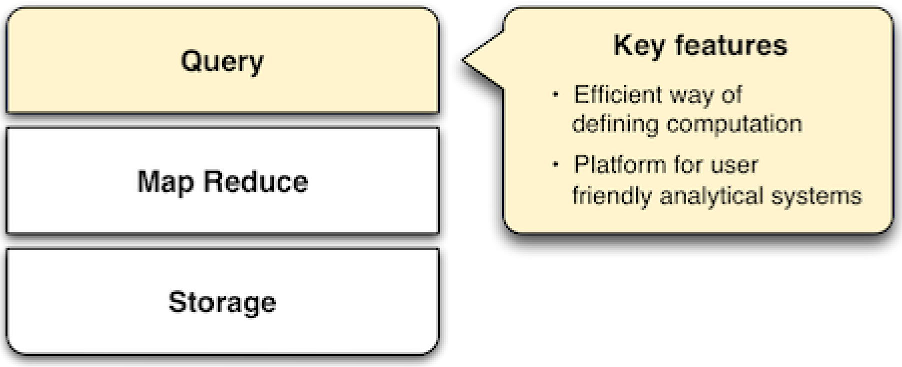
Other data warehousing solutions have opted to provide connectors with Hadoop, rather than integrating their own MapReduce functionality.

- Vertica, famously used by Farmville creator Zynga, is an MPP column-oriented database that offers a connector for Hadoop.
- Netezza is an established manufacturer of hardware data warehousing and analytical appliances. Recently acquired by IBM, Netezza is working with Hadoop distributor Cloudera to enhance the interoperation between their appliances and Hadoop. While it solves similar problems, Netezza falls outside of our SMAQ definition, lacking both the open source and commodity hardware aspects.

Although creating a Hadoop-based system can be done entirely with open source, it requires some effort to integrate such a system. Cloudera aims to make Hadoop enterprise-ready, and has created a unified Hadoop distribution in its Cloudera Distribution for Hadoop (CDH). CDH for Hadoop parallels the work of Red Hat or Ubuntu in creating Linux distributions. CDH comes in both a free edition and an Enterprise edition with additional proprietary components and support. CDH is an integrated and polished SMAQ environment, complete with user interfaces for operation and query. Cloudera's work has resulted in some significant contributions to the Hadoop open source ecosystem.

Query

Specifying MapReduce jobs in terms of defining distinct map and reduce functions in a programming language is unintuitive and inconvenient, as is evident from the Java code listings shown above. To mitigate this, SMAQ systems incorporate a higher-level query layer to simplify both the specification of the MapReduce operations and the retrieval of the result.


Query
Map Reduce
Storage
Key features
• Efficient way of defining computation
• Platform for user friendly analytical systems

Many organizations using Hadoop will have already written in-house layers on top of the MapReduce API to make its operation more convenient. Several of these have emerged either as open source projects or commercial products.

Query layers typically offer features that handle not only the specification of the computation, but the loading and saving of data and the orchestration of the processing on the MapReduce cluster. Search technology is often used to implement the final step in presenting the computed result back to the user.

Pig

Developed by Yahoo and now part of the Hadoop project, Pig provides a new high-level language, Pig Latin, for describing and running Hadoop MapReduce jobs. It is intended to make Hadoop accessible for developers familiar with data manipulation using SQL, and provides an interactive interface as well as a Java API. Pig integration is available for the Cassandra and HBase databases.

Below is shown the word-count example in Pig, including both the data loading and storing phases (the notation *$0* refers to the first field in a record).

```
input = LOAD 'input/sentences.txt' USING TextLoader();

words = FOREACH input GENERATE FLATTEN(TOKENIZE($0));

grouped = GROUP words BY $0;

counts = FOREACH grouped GENERATE group, COUNT(words);

ordered = ORDER counts BY $0;

STORE ordered INTO 'output/wordCount' USING PigStorage();
```

While Pig is very expressive, it is possible for developers to write custom steps in User Defined Functions (UDFs), in the same way that many SQL databases support the addition of custom functions. These UDFs are written in Java against the Pig API.

Though much simpler to understand and use than the MapReduce API, Pig suffers from the drawback of being yet another language to learn. It is SQL-like in some ways, but it is sufficiently different from SQL that it is difficult for users familiar with SQL to reuse their knowledge.

Hive

As introduced above, Hive is an open source data warehousing solution built on top of Hadoop. Created by Facebook, it offers a query language very similar to SQL, as well as a web interface that offers simple query-building functionality. As such, it is suited for non-developer users, who may have some familiarity with SQL.

Hive's particular strength is in offering ad-hoc querying of data, in contrast to the compilation requirement of Pig and Cascading. Hive is a natural starting point for more full-featured business intelligence systems, which offer a user-friendly interface for non-technical users.

The Cloudera Distribution for Hadoop integrates Hive, and provides a higher-level user interface through the HUE project, enabling users to submit queries and monitor the execution of Hadoop jobs.

Cascading, the API Approach

The Cascading project provides a wrapper around Hadoop's MapReduce API to make it more convenient to use from Java applications. It is an intentionally thin layer that makes the integration of MapReduce into a larger system more convenient. Cascading's features include:

- A data processing API that aids the simple definition of MapReduce jobs.
- An API that controls the execution of MapReduce jobs on a Hadoop cluster.
- Access via JVM-based scripting languages such as Jython, Groovy, or JRuby.
- Integration with data sources other than HDFS, including Amazon S3 and web servers.
- Validation mechanisms to enable the testing of MapReduce processes.

Cascading's key feature is that it lets developers assemble MapReduce operations as a flow, joining together a selection of “pipes”. It is well suited for integrating Hadoop into a larger system within an organization.

While Cascading itself doesn't provide a higher-level query language, a derivative open source project called Cascalog does just that. Using the Clojure JVM language, Cascalog implements a query language similar to that of Datalog. Though powerful and expressive, Cascalog is likely to remain a niche query language, as it offers neither the ready familiarity of Hive's SQL-like approach nor Pig's procedural expression. The listing below shows the word-count example in Cascalog: it is significantly terser, if less transparent.

```
(defmapcatop split [sentence]

        (seq (.split sentence "\\s+")))

(?<- (stdout) [?word ?count]

        (sentence ?s) (split ?s :> ?word)

        (c/count ?count))
```

Search with Solr

An important component of large-scale data deployments is retrieving and summarizing data. The addition of database layers such as HBase provides easier access to data, but does not provide sophisticated search capabilities.

To solve the search problem, the open source search and indexing platform Solr is often used alongside NoSQL database systems. Solr uses Lucene search technology to provide a self-contained search server product.

For example, consider a social network database where MapReduce is used to compute the influencing power of each person, according to some suitable metric. This ranking would then be reinjected to the database. Using Solr indexing allows operations on the social network, such as finding the most influential people whose interest profiles mention mobile phones, for instance.

Originally developed at CNET and now an Apache project, Solr has evolved from being just a text search engine to supporting faceted navigation and results clustering. Additionally, Solr can manage large data volumes over distributed servers. This makes it an ideal solution for result retrieval over big data sets, and a useful component for constructing business intelligence dashboards.

Conclusion

MapReduce, and Hadoop in particular, offers a powerful means of distributing computation among commodity servers. Combined with distributed storage and increasingly user-friendly query mechanisms, the resulting SMAQ architecture brings big data processing within reach for even small- and solo-development teams.

It is now economic to conduct extensive investigation into data, or create data products that rely on complex computations. The resulting explosion in capability has forever altered the landscape of analytics and data warehousing systems, lowering the bar to entry and fostering a new generation of products, services and organizational attitudes - a trend explored more broadly in Mike Loukides' "What is Data Science?" report.

The emergence of Linux gave power to the innovative developer with merely a small Linux server at their desk: SMAQ has the same potential to streamline data centers, foster innovation at the edges of an organization, and enable new startups to cheaply create data-driven businesses.

- Hadoop: The Definitive Guide
- Cassandra: The Definitive Guide
- Hands-on Cassandra
- REST in Practice
- REST
- MongoDB: The Definitive Guide