08 | Bigtable（一）：错失百亿的Friendster

2021-10-06 徐文浩

《大数据经典论文解读》 课程介绍 >

讲述：徐文浩
时长 18:52 大小 17.28M

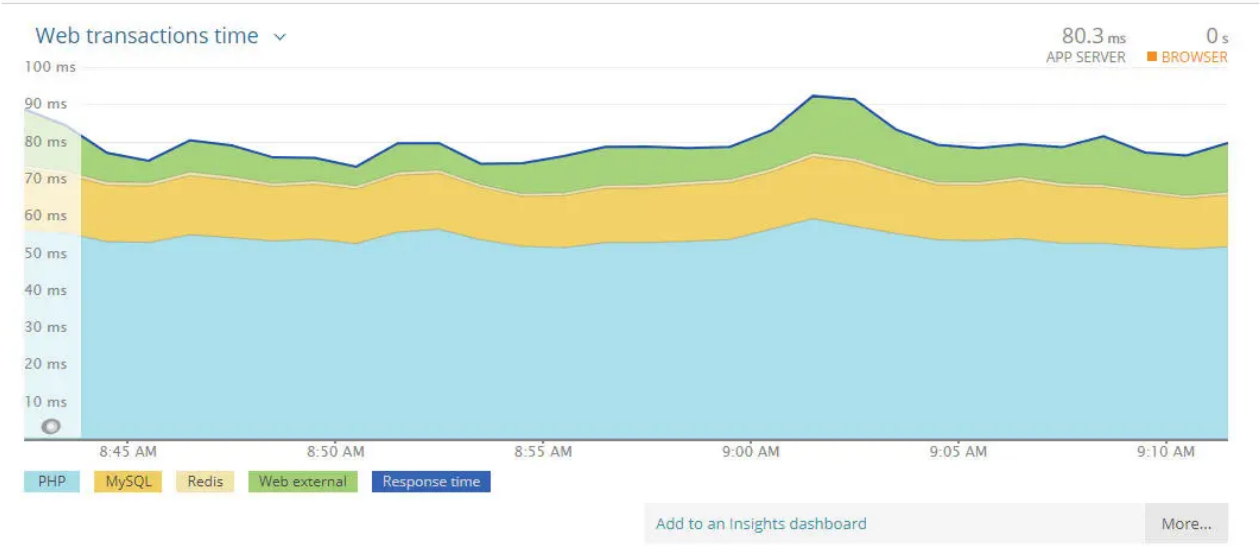
你好，我是徐文浩。

过去两周，我们一起看完了 GFS 和 MapReduce 的论文。相信这个时候的你一定自信满满，有一种“我上我也行”的感觉。的确，GFS 和 MapReduce 通过非常简单的设计，帮助我们解决了海量数据的存储、顺序写入，以及分布式批量处理的问题。

不过我们也要看到，GFS 和 MapReduce 的局限性也很大。

在 GFS 里，数据写入只对顺序写入有比较弱的一致性保障。而对于数据读取，虽然 G
支持随机读取，但是考虑到当时 Google 用的是孱弱的 5400 转的机械硬盘，实际上是支撑不了真正的高并发地读取的。

而 MapReduce，它也是一个批量处理数据的框架，吞吐量（throughput）确实很大，但是延时（latency）和额外开销（overhead）也不小。



性能可以分成“吞吐率”和“响应时间”两个角度，我在之前的专栏中单独讲过这个主题

> 可参考：《深入浅出计算机组成原理》

所以，即使有了 GFS 和 MapReduce，我们仍然有一个非常重要的需求没有在大型的分布式系统上得到满足，那就是可以高并发、保障一致性，并且支持随机读写数据的系统。而这个系统，就是接下来我们会深入讲解的 Bigtable。我会通过三讲，带你深入理解三个问题：

> Bigtable 想要解决什么问题？我们不能用 MySQL 这样的关系型数据库，搭建一个集群来解决吗？
>
> Bigtable 的架构是怎么样的？它是怎么来解决可用性、一致性以及容易运维这三个目标的？
>
> Bigtable 的底层数据结构是怎么样的？它是通过什么样的方式在机械硬盘上做到高并发地随机读写呢？

当你理解了这三个问题，相信你对分布式数据库的设计可以算是正式入门了，而且你对于计算机底层数据结构、硬件原理和大型系统设计之间的联系也建立起来了。这样，无论你后续是想专门从事分布式数据库的开发，还是成为一个熟知各类系统原理的架构师，都会有很大帮助。

那么，今天这节课，我们就来搞清楚第一个问题。

Bigtable 要解决什么问题？

不知道你有没有听说过Friendster这个网站？如果听说过的话，你多半暴露年龄了。Friendster 是一个成立于 2002 年的社交网络，比 Facebook 要早，更远远早于微信、微博乃至校内网。然而，Friendster 虽然起了个大早，却最终因为种种原因销声匿迹了。其中很重要的一个问题，就是在技术上，Friendster 没有解决好“**伸缩性**”（Scalability）问题。

从 2003 年开始，Friendster 就一直遇到严重的性能瓶颈，并且因为性能问题限制了很多功能的实现。甚至 MIT 在 2003 年的一门《Web 应用的软件工程》的课程里，还专门把 Friendster 的可用性分析，作为期中考试里的一题，可见当时 Friendster 的体验有多糟糕了。

我们可以一起来看看，当时 MIT 的学生们体验 Friendster 的一些评论：

From Amerson Li:

> Server speed is rather slow. This must be because of the great number of hits on the server. Sometimes, after I've read/replied to a message, it still appears on "Home" as having been unread. Also, whilst approving a testimonial, sometimes it doesn't move the testimonial into the "approved" section until I refresh about 5 mins later. These could be bugs? A possible fix, other than spending money buying faster machines is to provide a "low intensity" option for the site during peak hours of usage. This low intensity option could feature the following:

2003年Friendster体验的用户截图

> 服务器速度相当慢。这一定是因为服务器上有大量请求。有时候，在我阅读 / 回复了一条消息后，它仍然出现在“主页”上，显示为未读。

Friendster 在成立一年之后的 2003 年，就已经有 7500 万用户了，所以服务器压力的确很大。那么根据上面 MIT 学生的描述，我们可以想象一个简单的社交网络的功能，以及对应需要的读写请求数量：用户去看自己的时间线的时候，需要看到自己 150 个好友发的帖子。这里有两种解决办法。

一种是用户发帖子的时候，系统往所有好友的时间线里写一条数据，那么写入就会放大 150 倍。假设每天有 20% 的用户发 3 条帖子，那么写入的数据量就是：

> **7500 万 x 20% x 150 x 3 = 67.5 亿**

67.5 亿条随机数据写入，如果均匀分配到 10 个小时，每秒的随机写入量大概是：

> **67.5 亿 / (3600 秒 * 10) = 18.75 万次 / 秒**

还有一种是每个用户看自己时间线的时候，系统会查询 150 个好友各自发表的内容，然后做合并。那么对应的就是 22.5 亿次的随机数据读取，也就是每秒 6.25 万次的随机读取。

如果你读过我的《深入浅出计算机组成原理》中关于机械硬盘的那一讲，你就知道一块 7200 转的机械硬盘，只能支撑 100 的 IOPS，也就是 100 次随机读写。那按照上面的数据来看，我们至少需要 600 块硬盘，才能支撑简单地读取自己的时间线信息。事实上，600 块硬盘远远不够的，无论是读写什么数据，都不太可能只写入 1 条数据，更不可能只有 1 次随机读写，而我们的硬盘，也不可能刚好跑满 IOPS。

所以一方面，我们可能需要数千块硬盘，对应的，也就需要上千台服务器。另一方面，这个集群需要能够支持海量的随机读写，至少需要支持到每秒百万次级别的随机读写，而 Bigtable 就是这样一个系统。

“小数据”的 MySQL 集群

Bigtable 的论文发表于 2006 年，而基于论文实现的开源系统 HBase，要到 2008 年才第一次正式发布（0.18.0 版本）。所以，Friendster 并没有 Bigtable 可以用，在 2003 年，一家互联网公司面对“伸缩性”这个问题，最好的选择是使用一个 MySQL 集群。

维护一个几十乃至上百台服务器的 MySQL 集群是可行的，但是，如果要像 GFS 一样到一千乃至数千台服务器，还有可行性吗？下面我们就一起来看一下。

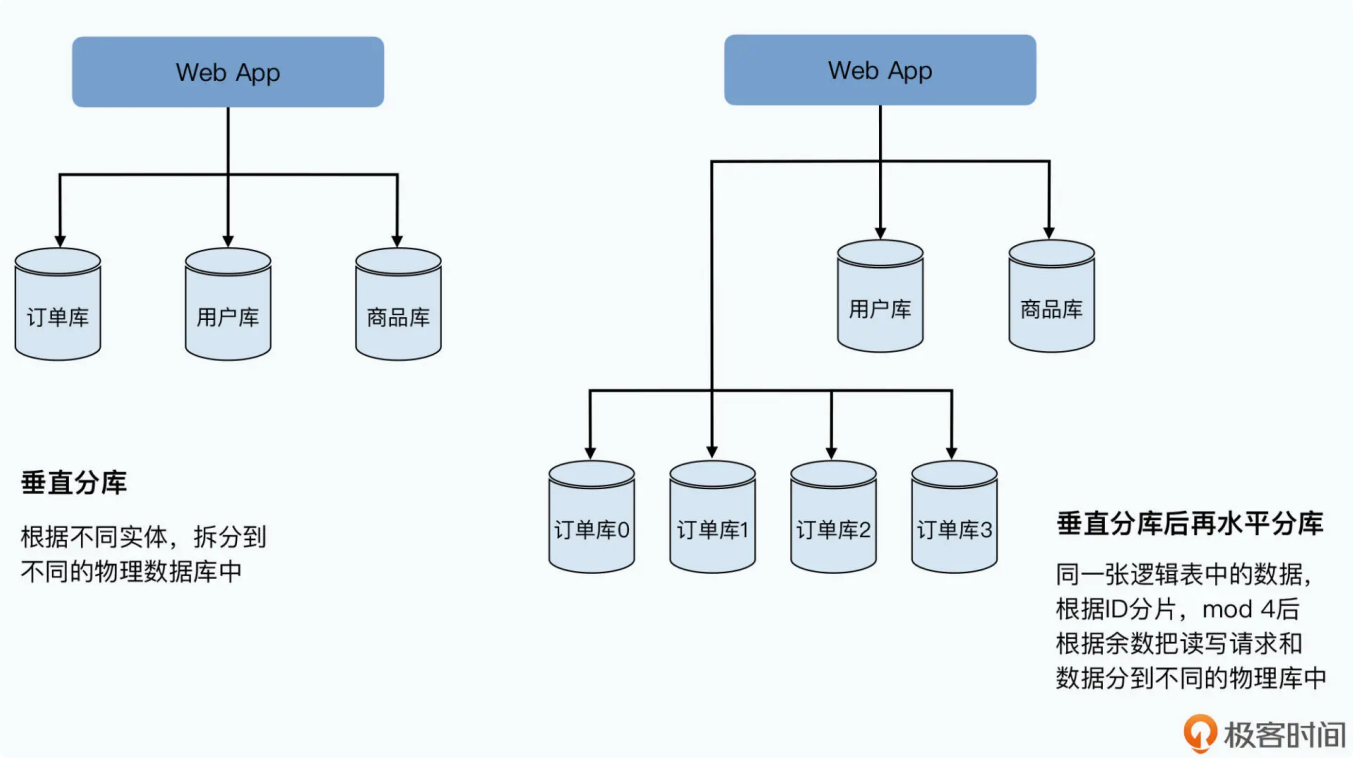
分库分表的扩容方式

一致性的随机读写，在单个服务器上似乎并不是什么问题。如果你是做后端应用开发的，肯定用过 MySQL 这样的数据库，你可以很容易地通过简单的 SQL，完成增删改查这样的

随机数据读写。如果要把单机的 MySQL 扩展成分布式，好像也不是什么难题，只要做个分库分表就好了，这些套路你应该会非常熟悉。

一般来说，我们会先做垂直分库，在电商的系统里，我们把用户、商品、订单的表拆分到不同服务器的数据库上。如果发现这样还不行，我们就再进行水平分库，把订单号 Hash 一下，然后取“模”（mod）个 4，拆分到 4 台不同的服务器的数据库里。

这样，每台机器只需要承接 1/4 的负担，看起来这种方式也能解决问题。当然，在分库分表的过程中，我们已经放弃了 MySQL 这样的关系型数据库的很多特性，比如外键关联这样的约束，以及单个数据库里面的跨行跨表的事务。



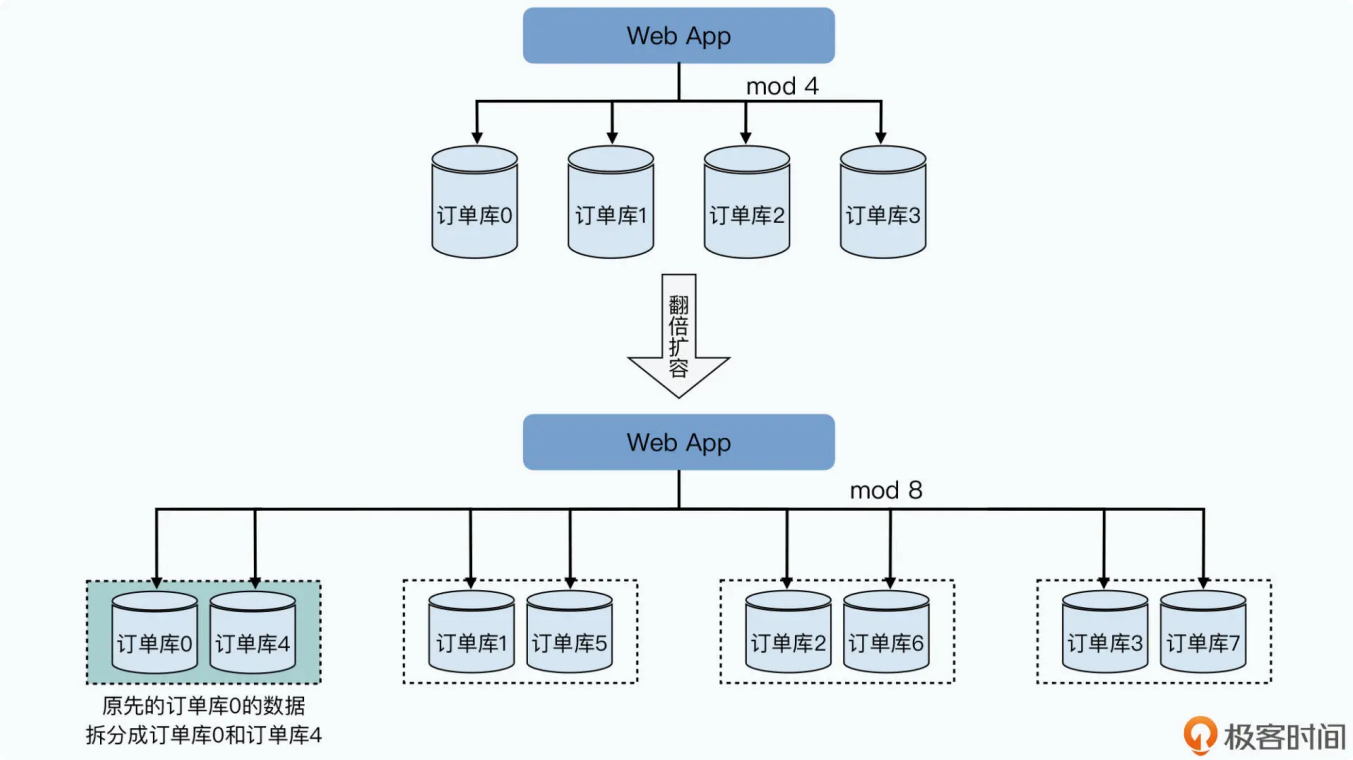
垂直和水平分库都是常见的解决单机数据库性能不足的解决方案

那么，为什么谷歌还需要发明一个 Bigtable 呢？这是因为分库分表，并不是一个很好的实现“可伸缩性”和“可运维性”的方案。基于分库分表的方案，运维起来会很费劲，主要体现在以下三点。

不得不进行的“翻倍扩容”

首先，是资源使用很浪费。当服务器性能出现瓶颈需要扩容的时候，我们常常只能采取“翻倍”分库增加服务器的方案。就以前面举的订单表为例，我们通过把订单号“模”上个 4，拆分到 4 个不同的服务器的数据库里。

而随着我们承接的订单越来越多，每天 SQL 查询的请求越来越多，服务器的峰值 CPU 可能超过了 60%。为了安全起见，我们希望对服务器进行扩容，让峰值 CPU 控制在 40% 以下。但是这个时候，我们没办法只是增加 4 * 0.6 / 0.4 - 4 = 2 台服务器，而是不得不“翻倍”增加 4 台服务器。



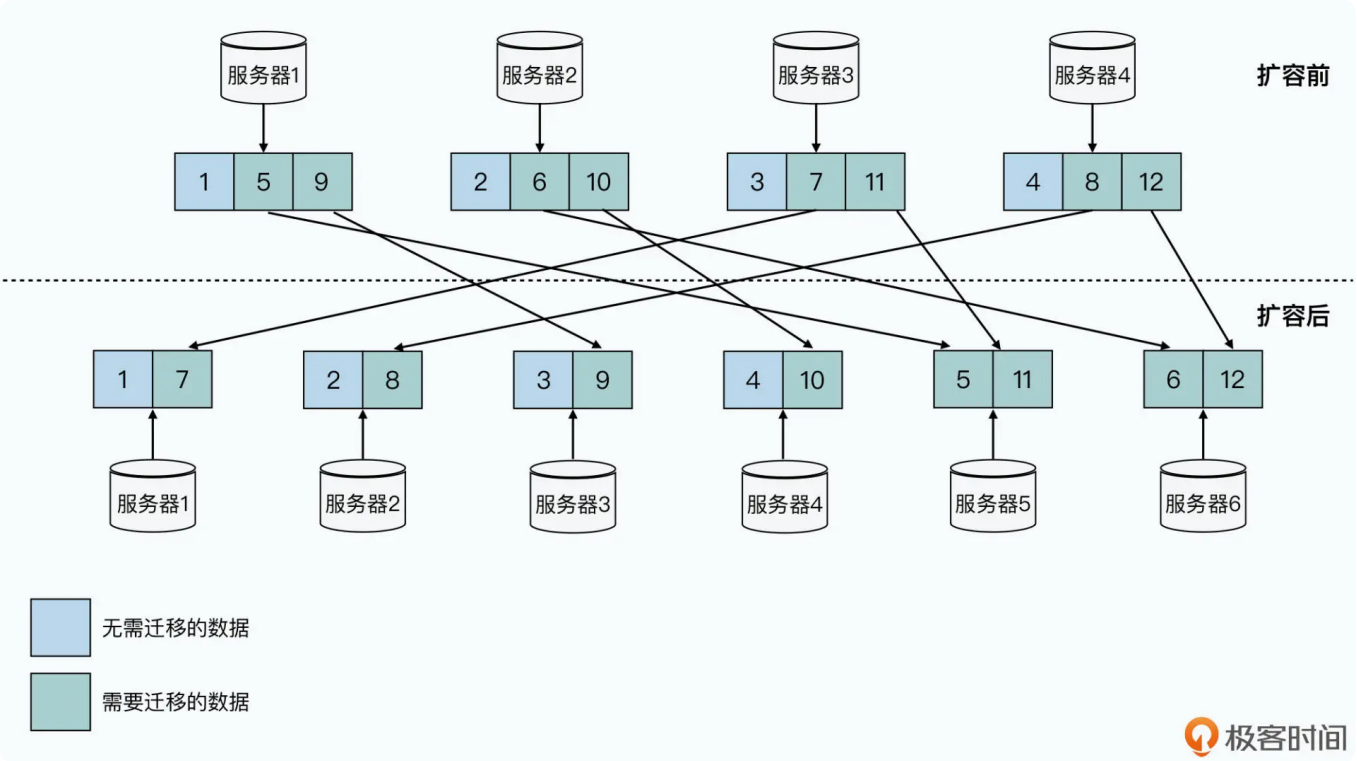
为什么呢？因为如果我们只增加 2 台服务器，把各个服务器的分片，从模上 4 变成模上 6，我们就需要在增加服务器之后，搬运大量的数据。并且这个数据搬运，不只是搬到新增加的服务器上，而是有些数据还要在原有的 4 台服务器上互相搬运。

这个搬运过程需要占用大量的网络带宽和硬盘读写，所以很有可能**要让数据库暂停服务**。而如果不暂停服务的话，我们就要面对在数据搬运的过程中，到底应该从哪个服务器读和写数据的问题，问题一下子就变得极其复杂了。

而翻倍扩容服务器，我们可以只需要简单复制 50% 的数据，并且在数据完成复制之后自动切换分片就可以了。但是**翻倍扩容的方案，自然就带来了很多浪费**，明明我们只需要加两台服务器，但是现在要加上四台。更浪费的是，我们增加的服务器，也许只是为了应对双十一促销这样的一小段时间，等到促销完成，我们又不再需要这些服务器了。

可这个时候，**如果我们需要缩减服务器，也会非常麻烦**，我们需要再把两台服务器的数据复制到一台服务器上，才能完成缩容。可以看到，这个集群虽然可以“伸缩”，但是伸缩起来非常不容易。

而我们希望的伸缩性是什么样的呢？自然是需要的时候，加 1 台服务器也加得，加 10 台服务器也加得。而用不上的时候，减少个 8 台 10 台服务器也没有问题，并且这些动作都不需要停机。这个，也是 Bigtable 的设计目标。



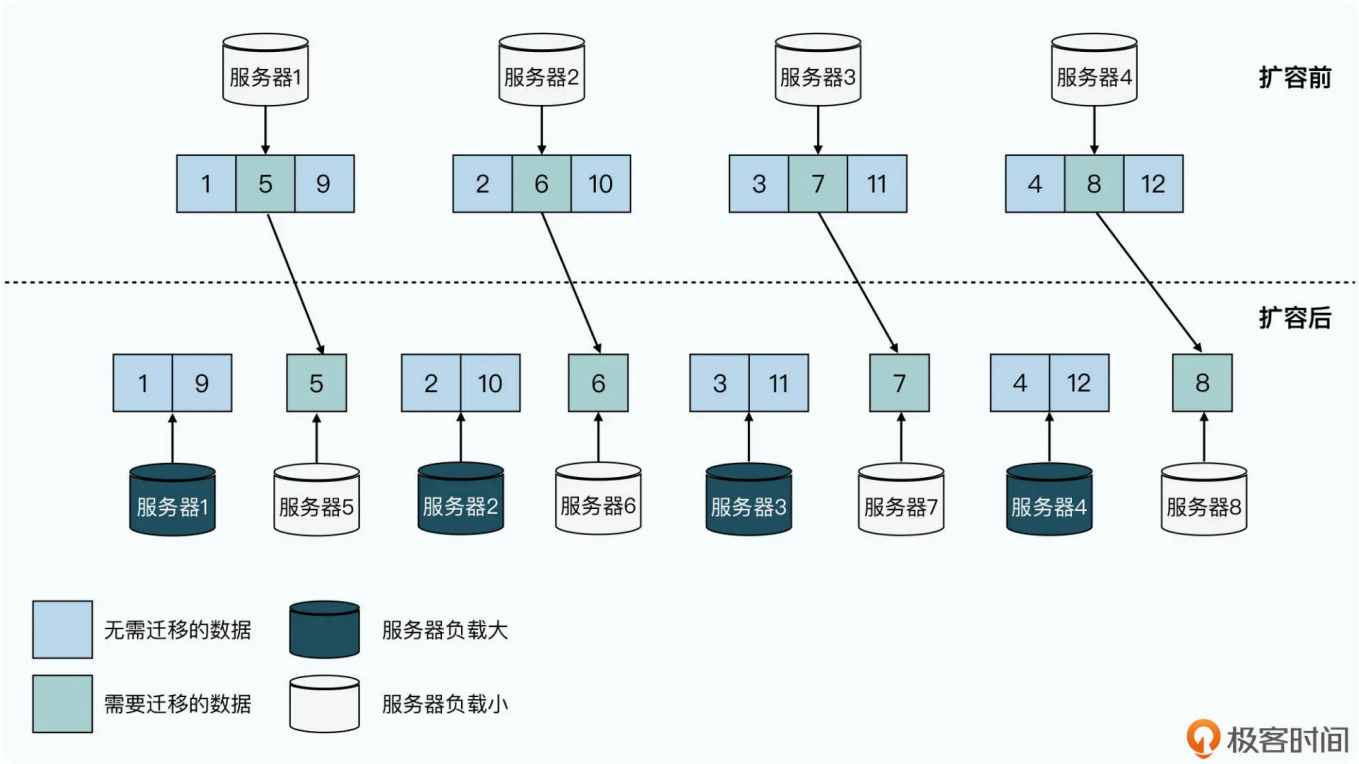
从4台扩容到6台服务器，如果我们还用mod N的方式，我们需要迁移2/3数据库中的数据

“我怎么早没想到”的数据分区

其次，是底层的数据分区策略对于应用不透明。如何分库和分表都需要开发人员来设计，撰写代码的时候，也常常要考虑底层分库分表的设计。

我们还是以 MySQL 分表作为例子，这一次我们来分一下用户表。我们还是分到 4 台机器上，用了用户出生的月份“模”上个 4。这个时候，很幸运，一年是有 12 个月，正好可以均匀分布到 4 台不同的机器上。

但是当我们进行扩容，变成 8 台机器之后，问题就出现了。我们会发现，服务器 A 分到了 1 月和 9 月生日的用户，而服务器 B 只分到了 6 月生日的用户。**在扩容之后，服务器 A 无论是数据量，还是日常读写的负载，都比服务器 B 要高上一倍。**而我们只能按照服务器 A 的负载要求来采购硬件，这也就意味着，服务器 B 的硬件性能很多都被浪费了。



即使是翻倍扩容，因为选择分片的字段问题，也会遇到分片不均匀带来的服务器负载差异问题

而且，不但用月份不行，用年份和日也不行。比如公司是 2018 年成立，2019 年和 2020 年快速成长，每年订单数涨 10 倍，如果你用年份来进行订单的分片，那么服务器之间的负载就要差上十倍。而用日的话，双十一这样的大促也会让你猝不及防。

你会发现，**使用 MySQL 集群，需要你在一开始就对如何切分数据做好精心设计**。一旦稍有不慎，设计上出现了数据倾斜，就很容易造成服务器忙得忙死，闲得闲死的现象。并且即使你已经考虑得非常仔细了，随着业务本身的变化，比如要搞个双十一，也会把你一朝打回原形。

那么，我们希望的分布式数据库是什么样的呢？自然是数据的分片是自适应的。比如 2019 年只有 100 万订单，那就分片到一个服务器节点上；2020 年有了 1000 万订单，自动给你分了 10 个节点；当 2021 年有 1 亿订单的时候，就给你分配上 100 个节点。而这一点，也同样是 Bigtable 的设计目标。

天天跑机房的人肉运维

最后，是故障恢复需要人工介入。在 MySQL 集群里，我们可以对每个服务器都准备一个高可用的备份，避免一出现故障整个集群就没法用了。但是此时，我们的运维人员仍然需要立刻介入，因为这个时候系统是**多了一个“单点”**的，我们需要手工添加一台新的服务器进入集群，同步到最新的数据。

我们可以一起来算一算，如果有一个 1000 台服务器的 MySQL 集群，每台服务器上都给插上 12 块硬盘，一共有 1 万 2 千块硬盘。这么多硬盘，我们到底要面临多少故障呢？

2003 年，谷歌的论文用的还是传统的机械硬盘，那个时候机械硬盘的可靠性数据我已经找不到了。不过我们可以看一下 2021 年的数据：Backblaze 这个公司从 2012 年开始就会发布硬盘的可靠性数据，从 2021 年 Q2 季度来看，他们数据中心里将近 18 万块的硬盘，在 90 天里一共坏了 439 块，差不多每天要坏上将近 5 块硬盘。

我们的 1 万 2 千块硬盘，是他们的 7% 不到，基本上 3 天也要坏上一块硬盘。要知道，这个还是只考虑了硬盘的硬件损坏，还没有算上 CPU、内存、交换机、网络等等各种各样的问题。

Backblaze Hard Drives Quarterly Failure Rates for Q2 2021

Reporting period: 4/1/2021 through 6/30/2021 inclusive

MFG	Model	Drive Size	Drive Count	Avg. Age (months	Drive Days	Drive Failures	AFR
HGST	HMS5C4040ALE640	4TB	3,209	62.1	291,656	5	0.63%
HGST	HMS5C4040BLE640	4TB	12,610	56.4	1,158,839	14	0.44%
Seagate	ST4000DM000	4TB	18,795	68.3	1,714,240	91	1.94%
Toshiba	MD04ABA400V	4TB	98	73.3	8,974	1	4.07%
Seagate	ST6000DX000	6TB	886	74.8	80,626	-	0.00%
HGST	HUH728080ALE600	8TB	1,077	40.7	98,004	1	0.37%
Seagate	ST8000DM002	8TB	9,733	56.9	885,873	33	1.36%
Seagate	ST8000NM0055	8TB	14,404	47.1	1,310,887	45	1.25%
Seagate	ST10000NM0086	10TB	1,199	44.5	109,134	3	1.00%
HGST	HUH721212ALE600	12TB	2,600	21.0	234,416	-	0.00%
HGST	HUH721212ALE604	12TB	10,754	4.5	789,373	3	0.14%
HGST	HUH721212ALN604	12TB	10,831	26.9	985,447	14	0.52%
Seagate	ST12000NM0007	12TB	3,552	31.7	726,571	41	2.06%
Seagate	ST12000NM0008	12TB	19,970	15.1	1,847,092	46	0.91%
Seagate	ST12000NM001G	12TB	10,557	9.2	894,875	10	0.41%
Seagate	ST14000NM001G	14TB	8,359	6.6	601,727	26	1.58%
Seagate	ST14000NM0138	14TB	1,653	6.8	151,236	23	5.55%
Toshiba	MG07ACA14TA	14TB	31,913	10.2	2,740,341	55	0.73%
Toshiba	MG07ACA14TEY	14TB	424	6.5	37,856	2	1.93%
WDC	WUH721414ALE6L4	14TB	8,400	6.8	751,725	10	0.49%
Seagate	ST16000NM001G	16TB	4,857	3.1	314,266	15	1.74%
Toshiba	MG08ACA16TEY	16TB	1,430	5.4	103,324	1	0.35%
WDC	WUH721816ALE6L0	16TB	624	3.0	47,963	-	0.00%
		Totals	**177,935**		**15,884,445**	**439**	**1.01%**

Backblaze

> 数据来源

那么，如果让我们的运维工程师，每个礼拜都有两天跑去数据中心换硬盘、运维系统，恐怕他别的事情也都干不了了。

而我们希望的可运维性是怎么样的呢？最好是 1000 台节点的服务器，坏个 10 台 8 台没事儿，系统能够自动把这 10 台 8 台服务器下线，用剩下的 990 台继续完成服务。我们的

运维人员只要 1 个月跑一趟机房批量换些机器就好，而不用 996 甚至 007 地担心硬件故障带来的不可用问题。

Bigtable 的设计目标

看到这里，相信你对 Bigtable 的设计目标应该更清楚了。最基础的目标自然是应对业务需求的，能够支撑**百万级别随机读写 IOPS**，并且伸缩到上千台服务器的一个数据库。但是光能撑起 IOPS 还不够。在这个数据量下，整个系统的“可伸缩性”和“可运维性”就变得非常重要。

这里的伸缩性，包括两点：

第一个，是**可以随时加减服务器，并且对添加减少服务器数量的限制要小**，能够做到忙的时候加几台服务器，过几个小时峰值过去了，就可以把服务器降下来。

第二个，是**数据的分片会自动根据负载调整**。某一个分片写入的数据多了，能够自动拆成多个分片来平衡负载。而如果负载大了，添加了服务器之后，也能很快平衡数据，让各个节点均匀承担压力。

而可运维性，则除了上面的两点之外，**小部分节点的故障，不应该影响整个集群的运行**，我们的运维人员也不用急匆匆地立刻去恢复。集群自身也要有很强的容错能力，能够把对应的请求和服务，调度到其他节点去。

那么，当我们回头看这个设计目标之后，会发现 Bigtable 的设计思路和 GFS 以及 MapReduce 一脉相承。

这三个系统的核心设计思路，就是把一个集群当成一台计算机。对于使用者来说，完全不用在意后面的分布式的存在。这样的设计思路，使得所有的工程师，并不需要学习什么新知识，只要熟悉这些分布式系统给到的接口，就能上手写大型系统。而这一点就让谷歌在很长一段时间都拥有极强的工程优势。

在 GFS+MapReduce+Bigtable 发布的前后几年里，谷歌发布了很多优秀的产品，比如 Gmail、Google Maps、Google Analytics 等，而这些产品的底层，就是优秀的分布式架构系统给谷歌带来的竞争优势。

当然，除了这些目标之外，Bigtable 也放弃了很多目标，其中有两个非常重要：

第一个是放弃了关系模型，也不支持 SQL 语言；

第二个，则是放弃了跨行事务，Bigtable 只支持单行的事务模型。

而这两个问题，一直要到 10 年后的 Spanner 里，才被真正解决好。在后续的课程里，你也会看到 Spanner 是怎么一步步从 Bigtable 进化而来的。到时候，你也可以对照着 Spanner 的论文来回头看看 Bigtable，看看这些逐步迭代的设计是否和你自己的思考和猜想一致。

小结

好了，相信到了这里，你对为什么我们需要一个 Bigtable，以及 Bigtable 的设计目标是什么就非常清楚了。**一个优秀的架构设计，是可以决定同样业务的两个公司的成败的。**其实，Friendster 在成立的第一年就收到了来自谷歌的 3000 万美元的收购邀请。如果有了 Bigtable 这样强健的底层数据架构，Friendster 不用一直到 2008 年都挣扎于糟糕的访问性能，即使不能战胜 Facebook，也至少有机会成为像 Twitter 这样数百亿美元的公司。

今天，我们一起从业务角度，看到了在 2003 年那个时间节点，工业界就已经有了每秒百万级别的随机读写的数据库的真实需求了。而仅仅是能通过堆硬件支撑起这样的 IOPS 还远远不够，我们还需要让系统很容易“伸缩”和“运维”。

无论是加减服务器、数据自动分片，还是硬件故障下的自动恢复，都不是一个“没有也能坚持，有了更好”的可选的需求。在“大数据时代”，在需要上千台服务器的集群之下，这些都变成了比优化一下性能、支持一下新的某个接口更重要的需求点了。

而 Bigtable 针对这些问题的答案，其实就是三点：

1. 第一点，是将整个系统的存储层，搭建在 GFS 上。然后通过单 Master 调度多 Tablets 的方式，使得整个集群非常容易伸缩和维护。
2. 第二点，是通过 MemTable+SSTable 这样一个底层文件格式，解决高速随机读写数据的问题。
3. 最后一点，则是通过 Chubby 这个高可用的分布式锁服务解决一致性的挑战。

接下来的两讲，我们还会深度解读 Bigtable 的实现。然后，我们会单独用一讲，来专门讲解 Chubby 这个分布式锁服务的论文。

希望通过这一讲，你能够学习到所有的架构设计都不是闭门造车，而是来自于真实的场景和真实的需求。而对于系统设计来说，除了业务需求和性能指标之外，有大量的隐性的需求同样重要甚至可能更加重要，Bigtable 特别重视的“可伸缩性”和“可运维性”就是最好的一个例子。

推荐阅读

今天我们聊了很多 MySQL 集群这样的关系型数据库集群在可伸缩性上会遇到的挑战。那是不是 MySQL 集群真的就只能搞个三五台服务器呢？倒也不是，战胜了 Friendster 的 Facebook 里，就有 MySQL 集群，有上千台服务器。

事实上，他们的工程师还在 2015 年的 SRECon 里面分享过这个主题“MySQL Automation at Facebook Scale”，我把链接放在了这里，你可以去看一看。不过看完你会发现，为了去运维这么大的 MySQL 集群，需要做的开发工作其实和实现一个 Bigtable 也差不多了。分布式锁、自动分片、自动故障隔离和恢复一个都少不了。

思考题

最后，给你留下一道思考题。我在一开始的 Friendster 的故事里，讲解了一个最简单的社交网络的业务模型。用户之间可以互相加好友，我们假设一旦加好友就是双向的（也就是不存在我是你的好友，但是你不是我的好友这样类似于微博的“单向关注”模型），然后好友发帖，你可以在自己的时间线里按照时间倒序看到。

那么，如果我们通过 Bigtable 来提供对应的数据服务，我们的 Bigtable 的表应该怎么设计呢？

欢迎给我留言，分享你的答案和见解，我们一起交流讨论。另外如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。

赞 1 提建议

上一篇 07 | MapReduce（二）：不怕失败的计算框架

精选留言 (2)

写留言

Monroe He
2021-10-06

Bigtable的row键是有序的，在生成row键的时候应该考虑将双向关注的用户尽量放在一起，可以考虑按照地区生成row键值，每个用户一条记录，将好友帖子存放在记录中，当好友row键值相邻时也有利于数据压缩

展开

峰
2021-10-06

思考题
假设采用方案一：一种是用户发帖子的时候，系统往所有好友的时间线里写一条数据。
关键在于通过主键避免读写热点，并减少过多的随机读写。
主键差不多就是： 时间+用户+random（比如0-9）
写的时候一方面用户较多，能够将写入压力分散到各个用户。且按时间排，数据的冷热...

展开