

由于自己正在做一个高性能大用户量的论坛程序，对高性能高并发服务器架构比较感兴趣，于是在网上收集了不少这方面的资料和大家分享。希望能和大家交流

msn: defender_ios@hotmail.com blog <http://www.benorz.org>

I	初创网站与开源软件	6
I	谈谈大型高负载网站服务器的优化心得!	8
I	Lighttpd+Squid+Apache 搭建高效率 Web 服务器	9
I	浏览量比较大的网站应该从哪几个方面入手?	17
I	用负载均衡技术建设高负载站点	20
I	大型网站的架构设计问题	25
I	原 开源平台的高并发集群思考	26
I	大型、高负载网站架构和应用初探 时间: 30-45 分钟	27
I	说说大型高并发高负载网站的系统架构	28
I	mixi 技术架构	51
	mixi.jp: 使用开源软件搭建的可扩展 SNS 网站	51
	总概关键点:	51
	1, Mysql 切分, 采用 Innodb 运行	52
	2, 动态 Cache 服务器 --	52
	美国 Facebok.com, 中国 Yeejee.com, 日本 mixi.jp 均采用开源分布式缓存服务器 Memcache	52
	3, 图片缓存和加	52
I	memcached+squid+apache deflate 解决网站大访问量问题	52
I	FeedBurner: 基于 MySQL 和 JAVA 的可扩展 Web 应用	53
I	YouTube 的架构扩展	55
I	了解一下 Technorati 的后台数据库架构	57
I	Myspace 架构历程	58
I	eBay 的数据量	64
I	eBay 的应用服务器规模	67
I	eBay 的数据库分布扩展架构	68
I	从 LiveJournal 后台发展看大规模网站性能优化方法	70
	一、LiveJournal 发展历程	70
	二、LiveJournal 架构现状概况	70
	三、从 LiveJournal 发展中学习	71
	1、一台服务器	71
	2、两台服务器	72
	3、四台服务器	73
	4、五台服务器	73
	5、更多服务器	74
	6、现在我们在哪里:	75
	7、现在我们在哪里	78
	8、现在我们在哪里	79
	9、缓存	80
	10、Web 访问负载均衡	80
	11、MogileFS	81

	Craigslist 的数据库架构.....	81
	Second Life 的数据拾零	82
	原 eBay 架构的思想金矿	84
	原 一天十亿次的访问—eBay 架构（一）	85
	七种缓存使用武器 为网站应用和访问加速发布时间:.....	92
	可缓存的 CMS 系统设计	93
	开发大型高负载类网站应用的几个要点[nightsailer]	105
	Memcached 和 Lucene 笔记	110
	使用开源软件，设计高性能可扩展网站	110
	面向高负载的架构 Lighttpd+PHP(FastCGI)+Memcached+Squid.....	113
	思考高并发高负载网站的系统架构.....	113
	"我在 SOHU 这几年做的一些门户级别的程序系统(C/C++开发)"	115
	原 中国顶级门户网站架构分析 1	116
	原 中国顶级门户网站架构分析 2	118
	服务器的大用户量的承载方案.....	120
	YouTube Scalability Talk.....	121
	High Performance Web Sites by Nate Koechley	123
	One dozen rules for faster pages	123
	Why talk about performance?.....	123
	Case Studies.....	124
	Conclusion.....	124
	Rules for High Performance Web Sites	124
	对于应用高并发，DB 千万级数量该如何设计系统哪？	125
	高性能服务器设计	130
	优势与应用：再谈 CDN 镜像加速技术	131
	除了程序设计优化，zend+ eacc(memcached)外，有什么办法能提高服务器的负载能力呢？ 135	
	如何规划您的大型 JAVA 多并发服务器程序	139
	如何架构一个“Just so so”的网站？	148
	最便宜的高负载网站架构.....	152
	负载均衡技术全攻略	154
	海量数据处理分析	164
	一个很有意义的 SQL 的优化过程（一个电子化支局中的大数据量的统计 SQL）	166
	如何优化大数据量模糊查询（架构，数据库设置，SQL..）	168
	求助:海量数据处理方法	169
	# re: 求助:海量数据处理方法 回复 更多评论	169
	海量数据库查询方略	169
	SQL Server 2005 对海量数据处理.....	170
	分表处理设计思想和实现.....	174
	Linux 系统高负载 MySQL 数据库彻底优化(1)	179
	大型数据库的设计与编程技巧 本人最近开发一个访问统计系统，日志非常的大，都保存在数据库里面。 我现在按照常规的设计方法对表进行设计，已经出现了查询非常缓慢地情形。 大家对于这种情况如何来设计数据库呢？把一个表分成多个表么？那么查询和插入数据库又有什么技巧呢？ 谢谢，村里面的兄弟们！	183

I	方案探讨,关于工程中数据库的问题. [已结贴].....	184
I	web 软件设计时考虑你的性能解决方案.....	190
I	大型 Java Web 系统服务器选型问题探讨	193
I	高并发高流量网站架构	210
	1.1 互联网的发展	210
	1.2 互联网网站建设的新趋势.....	210
	1.3 新浪播客的简介	211
	2.1 镜像网站技术	211
	2.2 CDN 内容分发网络	213
	2.3 应用层分布式设计.....	214
	2.4 网络层架构小结	214
	3.1 第四层交换简介	214
	3.2 硬件实现	215
	3.3 软件实现	215
I	网站架构的高性能和可扩展性.....	233
I	资料收集: 高并发 高性能 高扩展性 Web 2.0 站点架构设计及优化策略.....	243
I	CommunityServer 性能问题浅析	250
	鸡肋式的多站点支持	250
	内容数据的集中式存储	250
	过于依赖缓存	250
	CCS 的雪上加霜.....	250
	如何解决?	251
I	Digg PHP's Scalability and Performance.....	251
I	YouTube Architecture	253
	Information Sources	254
	Platform	254
	What's Inside?	254
	The Stats.....	254
	Recipe for handling rapid growth	255
	Web Servers	255
	Video Serving.....	256
	Serving Video Key Points	257
	Serving Thumbnails	257
	Databases	258
	Data Center Strategy	259
	Lessons Learned	260
	1. Jesse · Comments (78) · April 10th	261
	Library	266
	Friendster Architecture	273
	Information Sources.....	274
	Platform	274
	What's Inside?	274
	Lessons Learned.....	274
I	Feedblendr Architecture - Using EC2 to Scale	275

	The Platform	276
	The Stats	276
	The Architecture	276
	Lesson Learned	277
	Related Articles	278
	Comments	279
	Re: Feedblendr Architecture - Using EC2 to Scale.....	279
	Re: Feedblendr Architecture - Using EC2 to Scale.....	279
	Re: Feedblendr Architecture - Using EC2 to Scale.....	280
I	PlentyOfFish Architecture	281
	Information Sources.....	282
	The Platform	282
	The Stats	282
	What's Inside.....	283
	Lessons Learned	286
I	Wikimedia architecture	288
	Information Sources.....	288
	Platform	288
	The Stats	289
	The Architecture	289
	Lessons Learned	291
I	Scaling Early Stage Startups	292
	Information Sources.....	293
	The Platform	293
	The Architecture	293
	Lessons Learned	294
I	Database parallelism choices greatly impact scalability.....	295
I	Introduction to Distributed System Design	297
	Table of Contents	297
	Audience and Pre-Requisites.....	298
	The Basics.....	298
	So How Is It Done?	301
	Remote Procedure Calls	305
	Some Distributed Design Principles	307
	Exercises	308
	References	309
I	Flickr Architecture	309
	Information Sources.....	309
	Platform	310
	The Stats	310
	The Architecture	311
	Lessons Learned	316
	Comments	318
	How to store images?.....	318

	RE: How to store images?.....	318
I	Amazon Architecture	319
	Information Sources.....	319
	Platform	320
	The Stats	320
	The Architecture	320
	Lessons Learned	324
	Comments	329
	Jeff.. Bazos?	329
	Werner Vogels, the CTO of	329
	Re: Amazon Architecture	330
	Re: Amazon Architecture	330
	Re: Amazon Architecture	330
	It's WSDL	330
	Re: It's WSDL	331
	Re: Amazon Architecture	331
I	Scaling Twitter: Making Twitter 10000 Percent Faster.....	331
	Information Sources.....	332
	The Platform	332
	The Stats	333
	The Architecture	333
	Lessons Learned	336
	Related Articles.....	337
	Comments	338
	Re: Scaling Twitter: Making Twitter 10000 Percent Faster	338
	Re: Scaling Twitter: Making Twitter 10000 Percent Faster	338
	Re: Scaling Twitter: Making Twitter 10000 Percent Faster	338
	Re: Scaling Twitter: Making Twitter 10000 Percent Faster	339
	Re: Scaling Twitter: Making Twitter 10000 Percent Faster	339
	Re: Scaling Twitter: Making Twitter 10000 Percent Faster	339
	They could have been 20% better?	340
	Re: Scaling Twitter: Making Twitter 10000 Percent Faster	340
	Re: Scaling Twitter: Making Twitter 10000 Percent Faster	341
I	Google Architecture	341
	Information Sources.....	342
	Platform	342
	What's Inside?	342
	The Stats.....	342
	The Stack	343
	Reliable Storage Mechanism with GFS (Google File System)	343
	Do Something With the Data Using MapReduce.....	344
	Storing Structured Data in BigTable	346
	Hardware.....	347
	Misc	347

Future Directions for Google	348
Lessons Learned	348

不管怎么样，先要找出瓶颈在哪个部分：是 CPU 负荷太高（经常 100%），还是内存不够用（大量使用虚拟内存），还是磁盘 I/O 性能跟不上（硬盘指示灯狂闪）？这几个都是可以通过升级硬件来解决或者改善的（使用更高等级的 CPU，更快速和更大容量的内存，配置硬件磁盘阵列并使用更多数量的高速 SCSI 硬盘），但这需要较大的投入。

软件方面，如果使用了更大容量的内存和改善的 I/O 性能，已经能够大幅提高数据库的运行效率，还可以配置查询缓存和进一步优化数据库结构和查询语句，就能让数据库的性能再进一步。

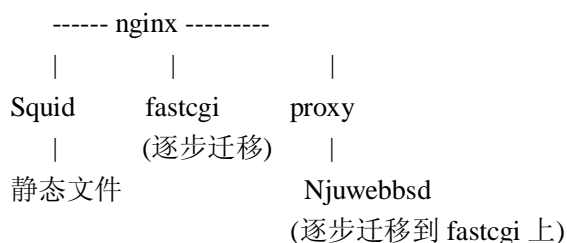
如果在服务器硬件投入上有困难，那就尽量生成静态页面。

作者: [BBSADM](#)

标 题: 目前的 web 系统架构

时 间: Fri Apr 6 20:15:56 2007

点 击: 100



最大好处是静态文件加速。

以后准备把帖子内容也静态化，实现最低负荷

而且用 nginx 做前台便于负载均衡，测试机可以拿来做静态文件的负载均衡

I 初创网站与开源软件

前面有一篇文章中提到过开源软件，不过主要是在系统运维的角度去讲的，主要分析一些系统级的开源软件(例如 bind, memcached)，这里我们讨论的是用于搭建初创网站应用的开源软件(例如 phpbb, phpparticle)，运行在 Linux, MySQL, Apache, PHP, Java 等下面。

创业期的网站往往采用比较简单的系统架构，或者是直接使用比较成熟的开源软件。使用开源软件的好处是搭建速度快，基本不需要开发，买个空间域名，下个软件一搭建，用个半天就搞定了，一个崭新的网站就开张了，在前期可以极大程度的节约时间成本和开发成本。

当然使用开源软件搭建应用也存在一些局限性，这是我们要重点研究的，而研究的目的是如何在开源软件选型时以及接下来的维护过程中尽量避免。

一方面是开源软件一般只有在比较成熟的领域才有，如果是一些创新型的项目很难找到合适的开源软件，这个时候没什么好的解决办法，如果非要用开源的话一般会找一个最相似的改一下。实际上目前开源的项目也比较多了，在 sf.net 上可以找到各种各样的开源项目。选型的时候尽量应该选取一个程序架构比较简单的，不一定越简单越好，但一定要简单，一目了然，别用什么太高级的特性，互联网应用项目不需要太复杂的框架。原因有两个，一个是框架复杂无非是为了实现更好的可扩展性和更清晰的层次，而我们正在做的互联网应用范围一般会比开源软件设计时所考虑的范围小的多，所以有的应用会显得设计过度，另外追求完美的层次划分导致的太复杂的继承派生关系也会影响到整个系统维护的工作量。建议应用只需要包含三个层就可以了，数据(实体)层，业务逻辑层，表现层。太复杂的设计容易降低开发效率，提高维护成本，在出现性能问题或者突发事件的时候也不容易找到原因。

另外一个问题是开源软件的后期维护和继续开发可能会存在问题，这一点不是绝对的，取决于开源软件的架构是否清晰合理，扩展性好，如果是较小的改动可能一般不存在什么问题，例如添加一项用户属性或者文章属性，但有些需求可能就不是很容易实现了。例如网站发展到一定阶段后可能会考虑扩展产品线，原来只提供一个论坛加上 `cms`，现在要再加上商城，那用户系统就会有问题，如何解决这个问题已经不仅仅是改一下论坛或者 `cms` 就可以解决了，这个时候我们需要上升到更高的层次来考虑问题，是否需要建立针对整个网站的用户认证系统，实现单点登录，用户可以在产品间无缝切换而且保持登录状态。由于网站初始的用户数据可能大部分都存放在论坛里，这个时候我们需要把用户数据独立出来就会碰到麻烦，如何既能把用户数据独立出来又不影响论坛原有系统的继续运行会是件很头痛的事情。经过一段时间的运行，除非是特别好的设计以及比较好的维护，一般都会在论坛里存在各种各样乱七八糟的对用户信息的调用，而且是直接针对数据库的，这样如果要移走的话要修改代码的工作量将不容忽视，而另外一个解决办法是复制一份用户数据出来，以新的用户数据库为主，论坛里的用户数据通过同步或异步的机制实现同步。最好的解决办法就是在选型时选一个数据层封装的比较好的，`sql` 代码不要到处飞的软件，然后在维护的时候保持系统原有的优良风格，把所有涉及到数据库的操作都放到数据层或者实体层里，这样无论对数据进行什么扩展，代码修改起来都比较方便，基本不会对上层的代码产生影响。

网站访问速度问题对初创网站来说一般考虑的比较少，买个空间或者托管服务器，搭建好应用后基本上就开始运转了，只有到真正面临极大的速度访问瓶颈后才会真正对这个问题产生重视。实际上在从网站的开始阶段开始，速度问题就会一直存在，并且会随着网站的发展也不断演进。一个网站最基本的要求，就是有比较快的访问速度，没有速度，再好的内容或服务也出不来。所以，访问速度在网站初创的时候就需要考虑，无论是采用开源软件还是自己开发都需要注意，数据层尽量能够正确，高效的使用 SQL。SQL 包含的语法比较复杂，实现同样一个效果如果考虑到应用层的不同实现方法，可能有好几种方法，但里面只有一种是最高效的，而通常情况下，高效的 SQL 一般是那个最简单的 SQL。在初期这个问题可能不是特别明显，当访问量大起来以后，这个可能成为最主要的性能瓶颈，各种杂乱无章的 SQL 会让人看的疯掉。当然前期没注意的话后期也有解决办法，只不过可能不会解决的特别彻底，但还是要吧非常有效的提升性能。看 MySQL 的 SlowQuery Log 是一个最为简便的方法，把执行时间超过 1 秒的查询记录下来，然后分析，把该加的索引加上，该简单的 SQL 简化。另外也可以通过 Showprocesslist 查看当前数据库服务器的死锁进程，从而锁定导致问题的 SQL 语句。另外在数据库配置文件上可以做一些优化，也可以很好的提升性能，这些文章在网站也比较多，这里就不展开。

这些工作都做了以后，下面数据库如果再出现性能问题就需要考虑多台服务器了，一台服务器已经解决不了问题了，我以前的文章中也提到过，这里也不再展开。

其它解决速度问题的办法就不仅仅是在应用里面就可以实现的了，需要从更高的高度去设计系统，考虑到服务器，网络的架构，以及各种系统级应用软件的配合，这里也不再展开。

良好设计并实现的应用+中间件+良好的分布式设计的数据库+良好的系统配置+良好的服务器/网络结构，就可以支撑起一个较大规模的网站了，加上前面的几篇文章，一个小网站发展到大网站的过程基本上就齐了。这个过程会是一个充满艰辛和乐趣的过程，也是一个可以逐渐过渡的过程，主动出击，提前考虑，减少救火可以让这个过程轻松一些。

I 谈谈大型高负载网站服务器的优化心得!

因为工作的关系，我做过几个大型网站（书库、证券）的相关优化工作，一般是在世界排行 1000-4000 以内的~~

这些网站使用的程序各不一样，配置也不尽相同，但是它们有一个共同的特点，

就是使用的是 FREEBSD 系统，高配置高负载，PV 值非常高，都是需要用两台以上独立主机来支持的网站~

我在优化及跟踪的过程中，开始效果也差强人意，也不太理想，后来通过阅读大量资料才慢慢理清了一些思路，写出来希望给大家有所帮助。

WEB 服务器配置是 DUAL XEON 2.4G 以上，2G 内存以上，SCSI 硬盘一块以上，FREEBSD 5.X 以上~~

数据库服务器与 WEB 服务器类似~~

书库程序是使用的 jieqi 的，论坛是使用的 Discuz! 的

apache 2.x + php 4.x + mysql 4.0.x + zend + 100M 光纤独享带宽

- 1、一定要重新编译内核，根据自己对内核认识的程度和服务器的具体配置来优化，记住打开 SMP，也可以使用 ULE 调度。
- 2、要优化系统的值，一般是添加入/etc/sysctl.conf 里面，要加大内核文件并发数量及其他优化等值。
- 3、APACHE 2 使用 perwork 工作模式就可以了，我试过 worker 模式，实在是差强人意呀。修改 httpd.conf 里面的值，加大并发数量和关闭不需要的模块。因为 apache 非常消耗内存，尽量轻装上阵~~ 可以适当的使用长连接。关闭日志。
- 4、PHP 编译的时候，注意要尽量以实用为目的加入参数，没有用到的坚决不加，以免浪费系统资源。
- 5、ZEND 要使用较小的优化等级，15 就足够了，1023 级别只会加重服务器负载~
- 6、MYSQL 要尽量少使用长连接，限制为 2-3 秒即可~
- 7、要全部采用手工编译方式，不要用 ports 安装，因为它会带上很多你不需要的模块，切记。
- 8、对于这类高负载高在线人数的大站，所有优化的思路就是把尽可能多的系统资源，提供给 WEB 和 MYSQL 服务，并且让这些服务单个进程可以占用尽可能少的系统资源。如果系统一开始大量使用 SWAP，对于这些服务器来说，服务器状态将会极剧恶化。
- 9、长时间观察跟踪调试，有什么问题尽快解决~~

就想到这些东东，欢迎大家补充~~

梦飞

<http://onlinecq.com>

2006/4/25

P.S. 补充我的几点优化：

- 1、编译 Apache PHP MySQL 时使用 GCC 参数传递对特定 CPU 进行优化；
- 2、如果网站小文件很多，可以考虑使用 reiserfs 磁盘系统，提升读写性能；
- 3、如不需要 .htaccess，则将 <Files .htaccess> 设置为 None

对于 apache 服务器繁忙，加大内存可以解决不少问题。

纯交互站点，mysql 性能会是一个瓶颈。需要长期跟踪更改参数。

I Lighttpd+Squid+Apache 搭建高效率 Web 服务器

架构原理

Apache 通常是开源界的首选 **Web** 服务器，因为它的强大和可靠，已经具有了品牌效应，可以适用于绝大部分的应用场合。但是它的强大有时候却显得笨重，配置文件得让人望而生畏，高并发情况下效率不太高。而轻量级的 **Web** 服务器 **Lighttpd** 却是后起之秀，其静态文件的响应能力远高于 **Apache**，据说是 **Apache** 的 2-3 倍。**Lighttpd** 的高性能和易用性，足以打动我们，在它能够胜任的领域，尽量用它。**Lighttpd** 对 **PHP** 的支持也很好，还可以通过 **Fastcgi** 方式支持其他的语言，比如 **Python**。

毕竟 **Lighttpd** 是轻量级的服务器，功能上不能跟 **Apache** 比，某些应用无法胜任。比如 **Lighttpd** 还不支持缓存，而现在的绝大部分站点都是用程序生成动态内容，没有缓存的话即使程序的效率再高也很难满足大访问量的需求，而且让程序不停的去做同一件事情也实在没有意义。首先，**Web** 程序是需要做缓存处理的，即把反复使用的数据做缓存。即使这样也还不够，单单是启动 **Web** 处理程序的代价就不少，缓存最后生成的静态页面是必不可少的。而做这个是 **Squid** 的强项，它本是做代理的，支持高效的缓存，可以用来给站点做反向代理加速。把 **Squid** 放在 **Apache** 或者 **Lighttpd** 的前端来缓存 **Web** 服务器生成的动态内容，而 **Web** 应用程序只需要适当地设置页面实效时间即可。

即使是大部分内容动态生成的网站，仍免不了会有一些静态元素，比如图片、**JS** 脚本、**CSS** 等等，将 **Squid** 放在 **Apache** 或者 **Lighttpd** 前端后，反而会使性能下降，毕竟处理 **HTTP** 请求是 **Web** 服务器的强项。而且已经存在于文件系统中的静态内容再在 **Squid** 中缓存一下，浪费内存和硬盘空间。因此可以考虑将 **Lighttpd** 再放在 **Squid** 的前面，构成 **Lighttpd+Squid+Apache** 的一条处理链，**Lighttpd** 在最前面，专门用来处理静态内容的请求，把动态内容请求通过 **proxy** 模块转发给 **Squid**，如果 **Squid** 中有该请求的内容且没有过期，则直接返回给 **Lighttpd**。新请求或者过期的页面请求交由 **Apache** 中 **Web** 程序来处理。经过 **Lighttpd** 和 **Squid** 的两级过滤，**Apache** 需要处理的请求将大大减少，减少了 **Web** 应用程序的压力。同时这样的构架，便于把不同的处理分散到多台计算机上进行，由 **Lighttpd** 在前面统一把关。

在这种架构下，每一级都是可以单独优化的，比如 **Lighttpd** 可以采用异步 **IO** 方式，**Squid** 可以启用内存来缓存，**Apache** 可以启用 **MPM** 等，并且每一级都可以使用多台机器来均衡负载，伸缩性很好。

实例讲解

下面以 **daviesliu.net** 和 **rainbud.net** 域下面的几个站点为例来介绍一下此方案的具体做法。

daviesliu.net 域下有几个用 **mod_python** 实现的 **blog** 站点，几个 **php** 的站点，一个 **mod_python** 的小程序，以后可能还会架设几个 **PHP** 和 **Django** 的站点。而服务器非常弱，CPU 为 **Celeron 500**，内存为 **PC 100 384M**，因此比较关注 **Web** 服务器的效率。这几个站点都是采用虚拟主机方式，开在同一台机器的同一个端口上。

Lighttpd 服务于 80 端口, Squid 运行在 3128 端口, Apache 运行在 81 端口。

Lighttpd 的配置

多个域名采用 `/var/www/domain/subdomain` 的目录结构, 用 `evhost` 模块配置 `document-root` 如下:

```
evhost.path-pattern      = var.basedir + "%0/%3/"
```

FtpSearch 中有 Perl 脚本, 需要启用 CGI 支持, 它是用来做 ftp 站内搜索的, 缓存的意义不大, 直接由 lighttpd 的 `mod_cgi` 处理:

```
$HTTP["url"] =~ "^/cgi-bin/" { # only allow cgi's in this directory
    dir-listing.activate = "disable" # disable directory listings
    cgi.assign = ( ".pl" => "/usr/bin/perl", ".cgi" => "/usr/bin/perl" )
}
```

bbs 使用的是 phpBB, 访问量不大, 可以放在 lighttpd(fastcgi)或者 apache(mod_php)下, 暂时使用 lighttpd, 设置所有.php 的页面请求有 fastcgi 处理:

```
fastcgi.server = ( ".php" => ( ( "host" => "127.0.0.1", "port"=>
1026, "bin-path" => "/usr/bin/php-cgi" ) ) )
```

blog.daviesliu.net 和 blog.rainbud.net 是用 mod_python 编写的 blogxp 程序, 所有静态内容都有扩展名, 而动态内容没有扩展名。blogxp 是用 python 程序生成 XML 格式的数据再交由 mod_xslt 转换成 HTML 页面, 只能放在 Apache 下运行。该站点采用典型

Lighttpd+Squid+Apache 方式处理:

```
$HTTP["host"] =~ "^blog" {
    $HTTP["url"] !~ "\." {
        proxy.server = ( "" => ( "localhost" => ( "host"=> "127.0.0.1", "port"=>
3128 ) ) ) #3128 端口为
    }
}
```

share 中有静态页面, 也有用 mod_python 处理的请求, 在/cgi/下:

```
$HTTP["host"] =~ "^share" {
    proxy.server = (
        "/cgi" => ( "localhost" => ( "host"=> "127.0.0.1", "port"=> 3128 ) )
    )
}
```

Squid 的配置

只允许本地访问:

```
http_port 3128
http_access allow localhost
http_access deny all
```

启用反向代理:

```
httpd_accel_host 127.0.0.1
httpd_accel_port 81          #apache 的端口
httpd_accel_single_host on
httpd_accel_with_proxy on    #启用缓存
httpd_accel_uses_host_header on #启用虚拟主机支持
```

此方向代理支持该主机上的所有域名。

Apache 的配置

配置/etc/conf.d/apache2, 让其加载 mod_python、mod_xslt、mod_php 模块:

```
APACHE2_OPTS="-D PYTHON -D XSLT -D PHP5"
```

所有网站的根目录:

```
<Directory "/var/www">
    AllowOverride All    #允许.htaccess 覆盖
    Order allow,deny
    Allow from all
</Directory>
```

基于域名的虚拟主机:

```
<VirtualHost *:81>
    ServerName blog.daviesliu.net
    DocumentRoot /var/www/daviesliu.net/blog
</VirtualHost>
```

这里明显没有 lighttpd 的 evhost 配置方便。

blog.daviesliu.net 下的.htaccess 设置(便于开发, 不用重启 Apache):

```
SetHandler mod_python
PythonHandler blogxp.publisher
PythonDebug On
PythonAutoReload On
```

```
<FilesMatch "\. ">
    SetHandler None    #静态文件直接由 Apache 处理
```

</FilesMatch>

<IfModule mod_xslt.c>

AddType text/xsl .xsl #防止对 xsl 文件进行转化

AddOutputFilterByType mod_xslt text/xml

XSLTCache off

XSLTProcess on

</IfModule>

Header set Pragma "cache"

Header set Cache-Control "cache"

在 blogxp.publisher 里面，还需要设置返回的文档类型和过期时间：

```
req.content_type = "text/xml"
```

```
req.headers_out['Expires'] = formatdate( time.time() + 60 * 5 )
```

经过这样的配置，所有站点都可以通过 80、3128、81 三个端口进行正常访问，80 端口用作对外的访问，以减少负荷。81 端口可以用作开发时的调试，没有缓存的困扰。

性能测试

由于时间和精力有限，下面只用 ab2 做一个并不规范的性能对比测试(每项都测多次取平均)，评价指标为每秒钟的请求数。

测试命令,以测试 lighttpd 上并发 10 个请求 scripts/prototype.js 为例:

ab2 -n 1000 -c 10 <http://blog.daviesliu.net:80/scripts/prototype.js>

静态内容: prototype.js (27kB)

Con	Lighttpd(:80)	Squid(:3128)	Apache(:81)
1	380	210	240
10	410	215	240
100	380	160	230

可见在静态内容上，Lighttpd 表现强劲，而 Squid 在没有配内存缓存的情况下比另两个 Web 服务器的性能要差些。

动态页面: /rss (31kB)

Con	Lighttpd(:80)	Squid(:3128)	Apache(:81)
1	103	210	6.17

10	110	200	6.04
100	100	100	6.24

在动态内容上，Squid 的作用非常明显，而 Lighttpd 受限于 Squid 的效率，并且还要低一大截。如果是有多台 Squid 来做均衡的话，Lighttpd 的功效才能发挥出来。
在单机且静态内容很少的情况下，可以不用 Lighttpd 而将 Squid 置于最前面。

14 Comments »

1. Re:Lighttpd+Squid+Apache 搭建高效率 Web 服务器

这种搭配倒是可以 不过正文描述有些地方有问题
light 可以自己加上 cache 支持 但从性能只考虑 cache 看比 squid 还好一点(平均每秒 3000+线上实际数据)
squid 那块说的不太对 处理静态优化到 99.99%以上的 hitratio 后 基本上作用非常大 对整体结构也很有好处
light+squid+apache 的结构过渡时期实际在线也跑过 当时是后端没做压缩支持 实际上每一块都可以根据自己需要 patch 没有最好 只有更合适 可管理性很重要

由 windtear 发表于 Wed Sep 13 13:38:15 2006

2. Re:Lighttpd+Squid+Apache 搭建高效率 Web 服务器

lighttpd + php 访问量大的话经常会导致 php 死掉，然后 500

不管是 local 还是 remote 方式

无奈，换 zeus 了，很坚挺，商业的就是商业的。

由 soff 发表于 Wed Sep 13 13:39:01 2006

3. Re:Lighttpd+Squid+Apache 搭建高效率 Web 服务器

His result looks weird, as a result, his conclusion is wrong.

Squid does not boost dynamic page at all, the speed gain in his test is because his client is requesting the same page in paralell, and squid will return the same page for the concurrent requests. I also guess that he did not configure expire time for static content in his web server, Squid will try to refetch the file with If-Modified-Since header for each request. That's why squid performs poor in the static test.

由 kxn 发表于 Wed Sep 13 13:41:24 2006

4. Re:Lighttpd+Squid+Apache 搭建高效率 Web 服务器

不太同意这一点，对 Squid 而言，动态页面和静态页面是一样的，只要设置好 HTTP 头，如果设置 Expires，是没有缓存效果的
如果不能 Cache 动态页面的话，那怎么起到加速效果？

由 *davies* 发表于 *Wed Sep 13 13:42:00 2006*

5. Re:Lighttpd+Squid+Apache 搭建高效率 Web 服务器

不好意思,英语不好,误导你了,上午在单位的机器没法输入中文
动态页面除非正确设置 HTTP 的过期时间头,否则就是没有加速效果的.反过来说,静态页面也需要设置过期时间头才对.

我说的设置 `expire` 时间是指的把过期时间设置到几分钟后或者几小时后,这样页面就在这段时间内完全缓冲在 `squid` 里面.

你实际测试动态页面有性能提升,这有几种可能,一是你的测试用的是并发请求同一个页面,`squid` 对并发的同页面请求,如果拿到的结果里面没有 `non cache` 头,会把这一个结果同时发回给所有请求,相当于有一个非常短时间的 `cache`,测试结果看起来会好很多,但是实际因为请求同一页面的机会不是很多,所以基本没啥改进,另一种情况是你用的动态页面程序是支持 `if-modified-since` 头的,他如果判断这个时间以后么有修改过,就直接返回 `not modified`,速度也会加快很多.

所以其实 `squid` 在实际生产中大部分时间都是用于缓冲静态页面的,动态页面不是不能缓冲,但是需要页面程序里面做很多配合,才能达到比较好的效果

`newsmth` 的 `www` 高峰时候是 600qps ,`squid` 端还是比较轻松,瓶颈在后端.

由 *kxn* 发表于 *Wed Sep 13 13:43:55 2006*

6. Re:Lighttpd+Squid+Apache 搭建高效率 Web 服务器

多谢你的详细解答!

我文章中写了,每个请求都会添加 `Expires` 头为当前时间的后 5 分钟,即每个页面的有效期为 5 分钟, `Squid` 似乎会根据这个时间来判断是否刷新缓存,无需服务器支持

`If-modified-since`

这个 5 分钟是根据页面的一般更新频率来确定的.

如果是访问量很大的 Web 应用,比如 `newsmth` 的 `www`, 如果将 `php` 页面的失效时间设置为 1-2 秒,则这段时间内的请求都会用缓存来回应,即使在这段缓存时间内数据更新了,但并不影响用户的使用,1-2 秒钟的滞后效应对用户的体验影响并不大,但换取的是更快的服务器响应尤其是访问量大但更新并不频繁的 `blog` 部分,这样做可能很有效

当然,如果实现了 `If-modified-since` 接口,将更有效,但工作量太大

由 *davies* 发表于 *Wed Sep 13 13:45:27 2006*

7. Re:Lighttpd+Squid+Apache 搭建高效率 Web 服务器

看来是我没有仔细看你文章了,确实没有注意到你文章里面提了 `expire` 头
静态页面也可以设置 `expire` 头的,用 `web server` 的一个模块就可以
这样基本就是全部用 `squid` 缓冲了.

没有 expire 头的时候,squid 就会每个请求都用 if modified since 去刷.

smthwww 的 php 页面 expire 时间是 5 分钟还是 10 分钟来着,我忘记了.

由 kxn 发表于 Wed Sep 13 13:46:46 2006

8. Re:Lighttpd+Squid+Apache 搭建高效率 Web 服务器

总的感觉多此一举阿, 如果没有非常巨大的访问量,squid 的解决方案就足够了。

如果真用了 lighttpd, 基本上没有什么必要要 apache 了,
除非是非常特别的应用, lighttpd 基本上都能支持的.

单机折腾这么多层, 是不会有性能收益的.

由 scanner 发表于 Wed Sep 13 13:48:44 2006

9. Re:Lighttpd+Squid+Apache 搭建高效率 Web 服务器

其实 lighttpd 的缓存功能很强大, 你可以看看他的 cml 文档, 能很好的解决动态内容的缓存问题。而且如果是单机服务器的话在架个 squid 意义不大。当然除非你要缓存的东西实在太多, squid 的 Bloom Filter 还是非常有效的。

由 Wei Litao [email] 发表于 Sat Sep 16 13:19:38 2006

10. Re: Lighttpd+Squid+Apache 搭建高效率 Web 服务器

lighttpd 有 bug, 内存泄漏比较严重。我现在用 nginx, 正在 lilybbs 上测试效果。其实把动态内容静态化才是最终出路。那些点击量真想去掉。

目前 lilybbs 的架构:

----- nginx -----

|||

Squid fastcgi proxy

| (逐步迁移) |

静态文件 Njuwebbsd

(逐步迁移到 fastcgi 上)

由 bianbian [email] [www] 发表于 Fri Apr 6 20:49:08 2007

11. Re: Lighttpd+Squid+Apache 搭建高效率 Web 服务器

ft.不支持空格排版。架构请看:

<http://bbs.nju.cn/blogcon?userid=BBSADM&file=1175861756>

由 bianbian [www] 发表于 Fri Apr 6 20:51:11 2007

12. Re: Lighttpd+Squid+Apache 搭建高效率 Web 服务器

另外。我觉得单机搞三层没什么必要, 你这个情况可以完全抛弃 apache。我现在的遗憾是 nginx 其他都很强, 就是 memcache 没完善, 所以必须弄个 Squid

由 bianbian [www] 发表于 Fri Apr 6 20:57:11 2007

13. Re: Lighttpd+Squid+Apache 搭建高效率 Web 服务器

我文中的那个方案只是在特殊场合才有用，呵呵

主来还是用来玩玩

点击其实可以通过分析 log 来离线做，或者单独放一些数据，用 ajax 来跟新这一部分，呵呵

由 *davies* 发表于 Sat Apr 7 01:31:18 2007

14. Re: Lighttpd+Squid+Apache 搭建高效率 Web 服务器

头一次听说 NginX，感觉应该是跟 lighttpd 同一个层次的东西，相差不会太大。如果要拼并发性能的话，估计平不过 yaws，改天做个简单测试。

由 *davies* 发表于 Sat Apr 7 01:39:14

I 浏览量比较大的网站应该从哪几个方面入手？

作者: 游戏人间 **时间:** 2007-6-15 04:23 PM **标题:** 浏览量比较大的网站应该从哪几个方面入手？

当然，提问前先将个人的一些理解分享。大家有的也请不吝共享，偶急切的需要这方面的经验....

下面所提到的主要是针对一般的网站，不包括下载或聊天室等特殊站点...

一、减少数据库的压力

缓存查询结果 / 建内存表

二、减少 Apache 的压力——减少 HTTP 的请求次数

背景图片全部做成一张然后用 CSS 控制位置/不使用 AJAX 来进行即时验证(不考虑客户体验什么的,通过拖长客户时间来减轻服务器压力)

三、减轻 I/O 压力

页面局部缓存

作者: 蟋蟀 **时间:** 2007-6-15 05:29 PM

咱也说点，只是理论，不知道对不对。

流量大的网站咱没做过。

一横向

1 \ 首先要考虑的就是硬件，适当的投入硬件，要比你搞那么多软件优化要实惠的多。

2 \ 在就是从 cpu 内存 硬盘 了。频繁操作的数据能存到内存中就存到内存中，能存到分布共享中就存储在分布共享内存中

其次考虑在考虑硬盘上。

二纵向

1 \ 从 web 的 http 的响应 应答考虑

web 要有服务器，所以如何优化服务器，如何通过配置服务器加速操作，能缓存的缓存，这方面的东西不少。

2、要是动态脚本，考虑使用的数据库 如何优化数据库、如何建立合理的表等操作 这方面细节同样不少

3、用 php 脚本，尽量少的 require 文件，毕竟每次 php 是一次性编译，而且每次到 require 都要返回 这个脚本方面的就要看程序员的水平了

还有很多

作者: fcicq 时间: 2007-6-19 09:30 PM

好久没出来了，难得碰上一篇可以回的帖子

一、减少数据库的压力

缓存查询结果 / 建内存表

有条件就把数据库尽量分开，减小数据库规模

杜绝超过 0.5s 的 queries - 非常重要！

开大内存索引

二、减少 Apache 的压力——减少 HTTP 的请求次数

背景图片全部做成一张然后用 CSS 控制位置/不使用 AJAX 来进行即时验证(不考虑客户体验什么的,通过拖长客户时间来减轻服务器压力)

背景图片? 这个没必要.

静态内容不要用 apache!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

三、减轻 I/O 压力

页面局部缓存

作者: Y 时间: 2007-6-20 11:59 AM

可以 lighttp+apache 配合的...lighttp 负责静态的如 image,js,css 等,apache 负责 php,用 rewrite 转发到 lighttp

甚至有研究表明,lighttp 处理 fastcgi 模式下的 php,要比 apache 等要快

性能上,lighttp 是要优于 apache 的,但稳定性就差点..

作者: sigmazel 时间: 2007-6-20 12:49 PM

WEB 方面:

1.脚本引用的资源文件如 css,js,image 可以多放几台服务器上,尽可能的压缩。

2.适当的加入 ajax

3.尽量控制 php 的代码行,如果方便的话,可以写成 com 或 so 级的

4.缓存

作者: php5 时间: 2007-6-20 06:55 PM

考虑硬件成本的话可以笼统地从以下着手

- 一、页面尽量静态化
- 二、配置服务器动态的走 **apache**,静态的走 **Lighttpd**
- 三、用最好的 OS 如 **FreeBSD**
- 四、重点优化 **mysql** 性能从编译、配置上入手
- 五、最基本的控制好程序性能及 **SQL** 查询
- 六、做缓存、做代理反向代理
- 七、页面上的优化了,节省流量上的考虑

作者: 奶瓶 时间: 2007-6-21 10:52 AM

静态文件用 **apache** 的代价很大,其实 **lighttpd** 和 **NGINX** 这类的也并不会小太多,有一些支持“文件至网卡”模式的特殊静态服务器可能划算一些。**php** 的调用文件个数可以做到比较精确的控制,**tmpfs** 一类的方法可以尝试,不要过分迷信 **memcached**,本地 **cache** 适当用用回保不错

作者: fengchen9127 时间: 2007-6-21 11:13 AM

优化数据库访问。

前台实现完全的静态化当然最好,可以完全不用访问数据库,不过对于频繁更新的网站,静态化往往不能满足某些功能。

缓存技术就是另一个解决方案,就是将动态数据存储在缓存文件中,动态网页直接调用这些文件,而不必再访问数据库,**WordPress** 和 **Z-Blog** 都大量使用这种缓存技术。我自己也写过一个 **Z-Blog** 的计数器插件,也是基于这样的原理。

如果确实无法避免对数据库的访问,那么可以尝试优化数据库的查询 **SQL**.避免使用 **Select * from** 这样的语句,每次查询只返回自己需要的结果,避免短时间内的大量 **SQL** 查询。

禁止外部的盗链。

外部网站的图片或者文件盗链往往会带来大量的负载压力,因此应该严格限制外部对于自身的图片或者文件盗链,好在目前可以简单地通过 **refer** 来控制盗链,**Apache** 自己就可以通过配置来禁止盗链,**IIS** 也有一些第三方的 **ISAPI** 可以实现同样的功能。当然,伪造 **refer** 也可以通过代码来实现盗链,不过目前蓄意伪造 **refer** 盗链的还不多,可以先不去考虑,或者使用非技术手段来解决,比如在图片上增加水印。

控制大文件的下载。

大文件的下载会占用很大的流量,并且对于非 **SCSI** 硬盘来说,大量文件下载会消耗 **CPU**,使得网站响应能力下降。因此,尽量不要提供超过 **2M** 的大文件下载,如果需要提供,建议将大文件放在另外一台服务器上。

使用不同主机分流主要流量

将文件放在不同的主机上,提供不同的镜像供用户下载。比如如果觉得 **RSS** 文件占用流量大,那么使用 **FeedBurner** 或者 **FeedSky** 等服务将 **RSS** 输出放在其他主机上,这样别人访问的流量压力就大多集中在 **FeedBurner** 的主机上,**RSS** 就不占用太多资源了。

使用流量分析统计软件。

在网站上安装一个流量分析统计软件,可以即时知道哪些地方耗费了大量流量,哪些页面需要再进行优化,因此,解决流量问题还需要进行精确的统计分析才可以。

I 用负载均衡技术建设高负载站点

转载自:IT.COM.CN | 2005 年 11 月 04 日 | 作者: | 浏览次数:57

Internet 的快速增长使多媒体网络服务器，特别是 Web 服务器，面对的访问者数量快速增加，网络服务器需要具备提供大量并发访问服务的能力。例如 Yahoo 每天会收到数百万次的访问请求，因此对于提供大负载 Web 服务的服务器来讲，CPU、I/O 处理能力很快会成为瓶颈。

简单的提高硬件性能并不能真正解决这个问题，因为单台服务器的性能总是有限的，一般来讲，一台 PC 服务器所能提供的并发访问处理能力大约为 1000 个，更为高档的专用服务器能够支持 3000-5000 个并发访问，这样的能力还是无法满足负载较大的网站的要求。尤其是网络请求具有突发性，当某些重大事件发生时，网络访问就会急剧上升，从而造成网络瓶颈，例如在网上发布的克林顿弹劾书就是很明显的例子。必须采用多台服务器提供网络服务，并将网络请求分配给这些服务器分担，才能提供处理大量并发服务的能力。

当使用多台服务器来分担负载的时候，最简单的办法是将不同的服务器用在不同的方面。按提供的内容进行分割时，可以将一台服务器用于提供新闻页面，而另一台用于提供游戏页面；或者可以按服务器的功能进行分割，将一台服务器用于提供静态页面访问，而另一些用于提供 CGI 等需要大量消耗资源的动态页面访问。然而由于网络访问的突发性，使得很难确定那些页面造成的负载太大，如果将服务的页面分割的过细就会造成很大浪费。事实上造成负载过大的页面常常是在变化中的，如果要经常按照负载变化来调整页面所在的服务器，那么势必对管理和维护造成极大的问题。因此这种分割方法只能是大方向的调整，对于大负载的网站，根本的解决办法还需要应用负载均衡技术。

负载均衡的思路下多台服务器为对称方式，每台服务器都具备等价的地位，都可以单独对外提供服务而无须其他服务器的辅助。然后通过某种负载分担技术，将外部发送来的请求均匀分配到对称结构中的某一台服务器上，而接收到请求的服务器都独立回应客户机的请求。由于建立内容完全一致的 Web 服务器并不复杂，可以使用服务器同步更新或者共享存储空间等方法来完成，因此负载均衡技术就成为建立一个高负载 Web 站点的关键性技术。

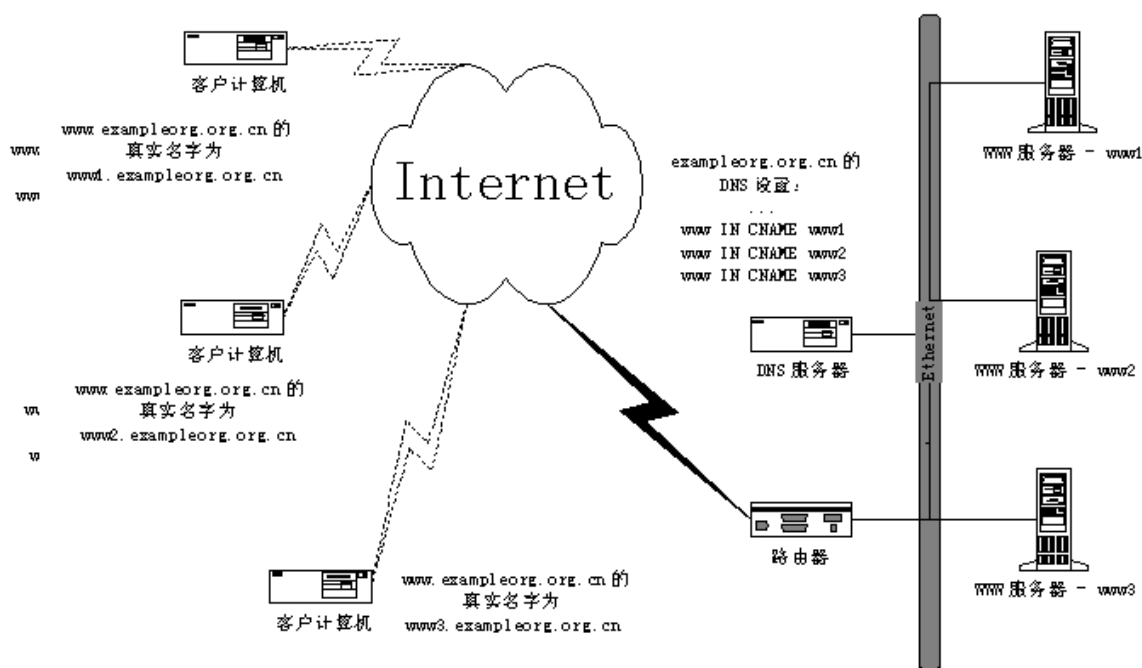
基于特定服务器软件的负载均衡

很多网络协议都支持“重定向”功能，例如在 HTTP 协议中支持 Location 指令，接收到这个指令的浏览器将自动重定向到 Location 指明的另一个 URL 上。由于发送 Location 指令比起执行服务请求，对 Web 服务器的负载要小的多，因此可以根据这个功能来设计一种负载均衡的服务器。任何时候 Web 服务器认为自己负载较大的时候，它就不再直接发送回浏览器请求的网页，而是送回一个 Location 指令，让浏览器去服务器集群中的其他服务器上获得所需要的网页。

在这种方式下，服务器本身必须支持这种功能，然而具体实现起来却有很多困难，例如一台服务器如何能保证它重定向过的服务器是比较空闲的，并且不会再次发送 Location 指令？Location 指令和浏览器都没有这方面的支持能力，这样很容易在浏览器上形成一种死循环。因此这种方式实际应用当中并不多见，使用这种方式实现的服务器集群软件也较少。有些特定情况下可以使用 CGI（包括使用 FastCGI 或 mod_perl 扩展来改善性能）来模拟这种方式去分担负载，而 Web 服务器仍然保持简洁、高效的特性，此时避免 Location 循环的任务将由用户的 CGI 程序来承担。

基于 DNS 的负载均衡

由于基于服务器软件的负载均衡需要改动软件，因此常常是得不偿失，负载均衡最好是在服务器软件之外来完成，这样才能利用现有服务器软件的种种优势。最早的负载均衡技术是通过 DNS 服务中的随机名字解析来实现的，在 DNS 服务器中，可以为多个不同的地址配置同一个名字，而最终查询这个名字的客户机将在解析这个名字时得到其中的一个地址。因此，对于同一个名字，不同的客户机会得到不同的地址，他们也就访问不同地址上的 Web 服务器，从而达到负载均衡的目的。



例如如果希望使用三个 Web 服务器来回应对 www.exampleorg.org.cn 的 HTTP 请求，就可以设置该域的 DNS 服务器中关于该域的数据包括有与下面例子类似的结果：

```
www1 IN A 192.168.1.1  
www2 IN A 192.168.1.2  
www3 IN A 192.168.1.3  
www IN CNAME www1  
www IN CNAME www2  
www IN CNAME www3
```

此后外部的客户机就可能随机的得到对应 www 的不同地址，那么随后的 HTTP 请求也就发送给不同地址了。

DNS 负载均衡的优点是简单、易行，并且服务器可以位于互联网的任意位置上，当前使用在包括 Yahoo 在内的 Web 站点上。然而它也存在不少缺点，一个缺点是为了保证 DNS 数据及时更新，一般都要将 DNS 的刷新时间设置的较小，但太小就会造成太大的额外网络流量，并且更改了 DNS 数据之后也不能立即生效；第二点是 DNS 负载均衡无法得知服务器之间的差异，它不能做到为性能较好的服务器多分配请求，也不能了解到服务器的当前状态，甚至会出现客户请求集中在某一台

服务器上的偶然情况。

反向代理负载均衡

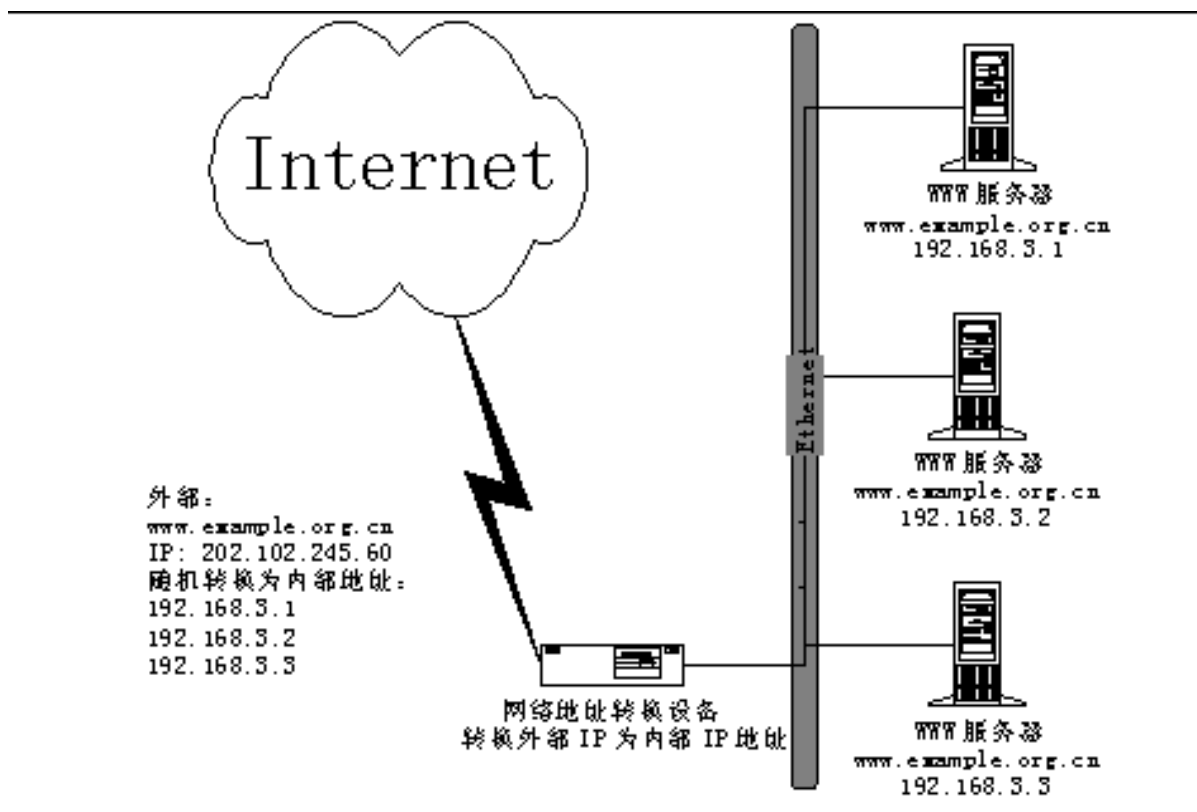
使用代理服务器可以将请求转发给内部的 Web 服务器，使用这种加速模式显然可以提升静态网页的访问速度。因此也可以考虑使用这种技术，让代理服务器将请求均匀转发给多台内部 Web 服务器之一上，从而达到负载均衡的目的。这种代理方式与普通的代理方式有所不同，标准代理方式是客户端使用代理访问多个外部 Web 服务器，而这种代理方式是多个客户端使用它访问内部 Web 服务器，因此也被称为反向代理模式。

实现这个反向代理能力并不能算是一个特别复杂的任务，但是在负载均衡中要求特别高的效率，这样实现起来就不是十分简单的了。每针对一次代理，代理服务器就必须打开两个连接，一个为对外的连接，一个为对内的连接，因此对于连接请求数量非常大的时候，代理服务器的负载也就非常之大了，在最后反向代理服务器会成为服务的瓶颈。例如，使用 Apache 的 mod_rproxy 模块来实现负载均衡功能时，提供的并发连接数量受 Apache 本身的并发连接数量的限制。一般来讲，可以使用它来对连接数量不是特别大，但每次连接都需要消耗大量处理资源的站点进行负载均衡，例如搜寻。

使用反向代理的好处是，可以将负载均衡和代理服务器的高速缓存技术结合在一起，提供有益的性能，具备额外的安全性，外部客户端不能直接访问真实的服务器。并且实现起来可以实现较好的负载均衡策略，将负载可以非常均衡的分给内部服务器，不会出现负载集中到某个服务器的偶然现象。

基于 NAT 的负载均衡技术

网络地址转换为在内部地址和外部地址之间进行转换，以便具备内部地址的计算机能访问外部网络，而当外部网络中的计算机访问地址转换网关拥有的某一外部地址时，地址转换网关能将其转发到一个映射的内部地址上。因此如果地址转换网关能将每个连接均匀转换为不同的内部服务器地址，此后外部网络中的计算机就各自与自己转换得到的地址上服务器进行通信，从而达到负载分担的目的。



地址转换可以通过软件方式来实现，也可以通过硬件方式来实现。使用硬件方式进行操作一般称为交换，而当交换必须保存 TCP 连接信息的时候，这种针对 OSI 网络层的操作就被称为第四层交换。支持负载均衡的网络地址转换为第四层交换机的一种重要功能，由于它基于定制的硬件芯片，因此其性能非常优秀，很多交换机声称具备 400MB-800MB 的第四层交换能力，然而也有一些资料表明，在如此快的速度下，大部分交换机就不再具备第四层交换能力了，而仅仅支持第三层甚至第二层交换。

然而对于大部分站点来讲，当前负载均衡主要是解决 Web 服务器处理能力瓶颈的，而非网络传输能力，很多站点的互联网连接带宽总共也不过 10MB，只有极少的站点能够拥有较高速的网络连接，因此一般没有必要使用这些负载均衡器这样的昂贵设备。

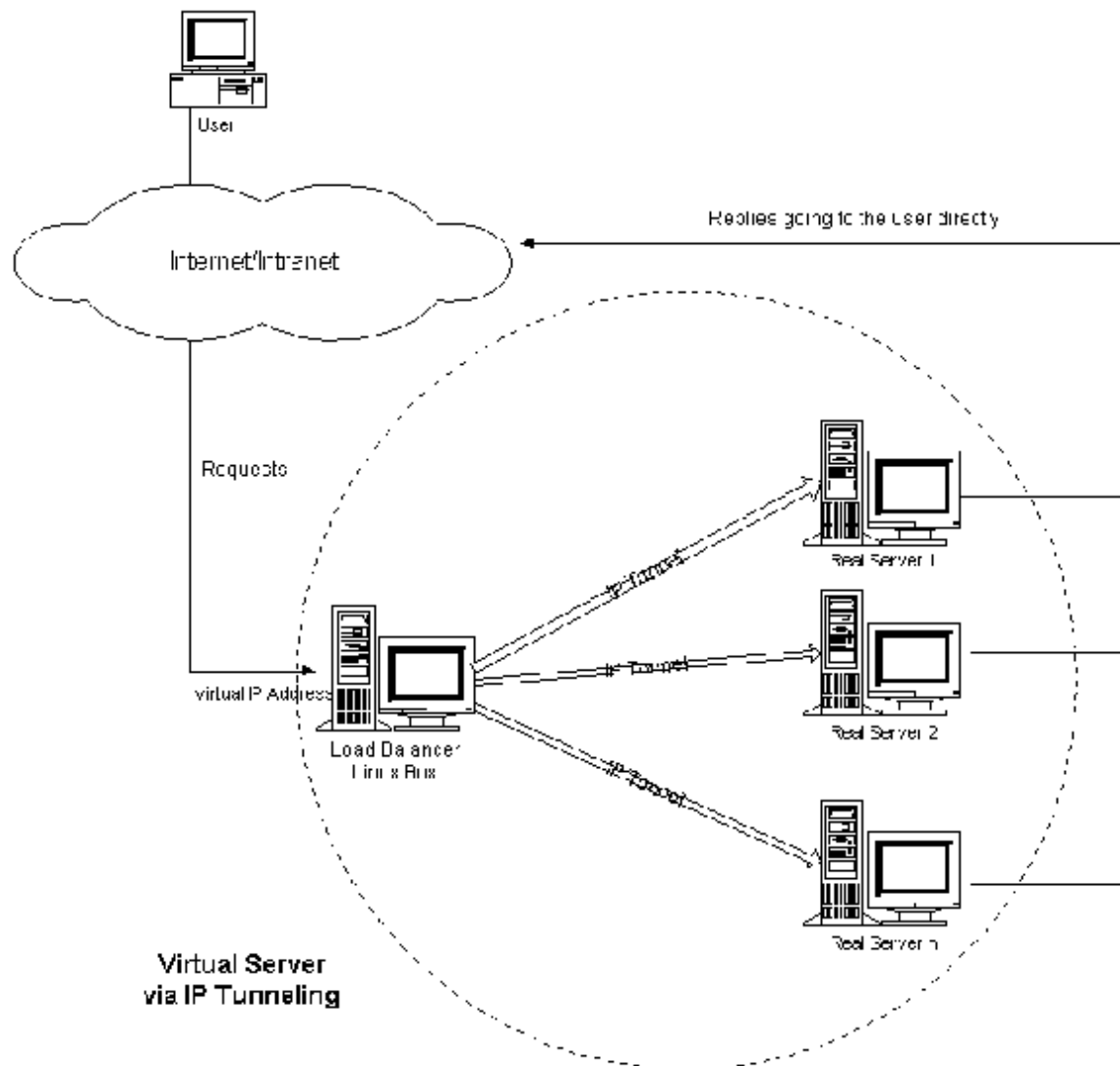
使用软件方式来实现基于网络地址转换的负载均衡则要实际的多，除了一些厂商提供的解决方法之外，更有效的方法是使用免费的自由软件来完成这项任务。其中包括 Linux Virtual Server Project 中的 NAT 实现方式，或者本文作者在 FreeBSD 下对 natd 的修订版本。一般来讲，使用这种软件方式来实现地址转换，中心负载均衡器存在带宽限制，在 100MB 的快速以太网条件下，能得到最快达 80MB 的带宽，然而在实际应用中，可能只有 40MB-60MB 的可用带宽。

扩展的负载均衡技术

上面使用网络地址转换来实现负载分担，毫无疑问所有的网络连接都必须通过中心负载均衡器，那么如果负载特别大，以至于后台的服务器数量不再是在是几台、十几台，而是上百台甚至更多，即便是使用性能优秀的硬件交换机也回遇到瓶颈。此时问题将转变为，如何将那么多台服务器分布到各个互联网的多个位置，分散网络负担。当然这可以通过综合使用 DNS 和 NAT 两种方法来实现，

然而更好的方式是使用一种半中心的负载均衡方式。

在这种半中心的负载均衡方式下，即当客户请求发送给负载均衡器的时候，中心负载均衡器将请求打包并发送给某个服务器，而服务器的回应请求不再返回给中心负载均衡器，而是直接返回给客户，因此中心负载均衡器只负责接受并转发请求，其网络负担就较小了。



上图来自 Linux Virtual Server Project, 为他们使用 IP 隧道实现的这种负载分担能力的请求/回应过程，此时每个后台服务器都需要进行特别的地址转换，以欺骗浏览器客户，认为它的回应为正确的回应。

同样，这种方式的硬件实现方式也非常昂贵，但是会根据厂商的不同，具备不同的特殊功能，例如对 SSL 的支持等。

由于这种方式比较复杂，因此实现起来比较困难，它的起点也很高，当前情况下网站并不需要这么大的处理能力。

比较上面的负载均衡方式，DNS 最容易，也最常用，能够满足一般的需求。但如果需要进一步的管理和控制，可以选用反向代理方式或 NAT 方式，这两种之间进行选择主要依赖缓冲是不是很重要，最大的并发访问数量是多少等条件。而如果网站上对负载影响很厉害的 CGI 程序是由网站自己开发的，也可以考虑在程序中自己使用 `Locaction` 来支持负载均衡。半中心化的负载分担方式至少在国内当前的情况下还不需要。

I 大型网站的架构设计问题

在 CSDN 上看到一篇文章（<http://blog.csdn.net/fww80/archive/2006/04/28/695293.aspx>）讨论大型高并发负载网站的系统架构问题，作者提出了几点建议：

1. HTML 静态化，这可以通过 CMS 自动实现；
2. 图片服务器分离（类似的，在视频网站中，视频文件也应独立出来）；
3. 数据库集群和库表散列，Oracle、MySQL 等 DBMS 都有完美的支持；
4. 缓存，比如使用 Apache 的 Squid 模块，或者是开发语言的缓存模块，；
5. 网站镜像；
6. 负载均衡。

作者将负载均衡称为“是大型网站解决高负荷访问和大量并发请求采用的终极解决办法”，并提出“一个典型的使用负载均衡的策略就是，在软件或者硬件四层交换的基础上搭建 squid 集群”。在实践时可以考虑建立应用服务器集群和 Web 服务器集群，应用服务器集群可以采用 Apache+Tomcat 集群和 WebLogic 集群等，Web 服务器集群可以用反向代理，也可以用 NAT 的方式，或者多域名解析均可。

从提升网站性能的角度出发，静态资源不应和应用服务器放在一起，数据库服务器也应尽量独立开来。在典型的 MVC 模式中，由谁来完成数据逻辑处理的，对系统性能有着至关重要的影响。以 Java EE 为例，在 OO 的设计思想中，我们强调系统的抽象、重用、可维护性，强调下层的更改不会扩散到上层逻辑，强调系统移植的便捷性，因而往往存在一种过分抽象的问题，比如在 Hibernate 的基础上再加入一层 DAO 的设计。另外一方面，却会忽视利用 DBMS 本身的优秀特性（存储过程、触发器）来完成高效的数据处理。诚然，如果客户要求将数据从 Oracle 移植到 MySQL，那么 DBMS 特性的东西越少，移植便越容易。但事实上，在实践中，提出类似移植要求的情况非常少见，因此在做架构设计时，不一定为了这种潜在的需求而大幅牺牲系统的性能与稳定性。此外，我不建议采用分布式数据库管理结构，这样带来的开销太大，数据维护也是个头痛的问题，尽可能采用集中式的数据管理。

在商业系统中，算法逻辑本身并不复杂，在这种情况下，程序设计本身的好坏不会对系统的性能造成致命的影响。重要的影响因素反而变为软件系统架构本身。在传统的 CORBA、J2EE、DCOM 等对象模型中，我们看到专家们对分布式对象计算的理论偏好，但实践证明，对象的分布带来的恶劣影响远远胜过其积极意义。这也是现在轻量级的开发框架受推崇的一个重要原因。如果

能用简单的，就不要用复杂的，例如能够用 Python、RoR 完成的任务，是否一定要用 Java 来做？我看未必。对于用户来说，他们关心的不是采用什么先进的技术，而是我们提供的产品能否满足他的需求。而且，Python、RoR 这些开发工具已经强大到足以应对大部分网站应用，在各种缓存系统的帮助下，在其他技术的协调配合下，完全能够胜任高负载高并发的网站访问任务。

在 HTML 静态化方面，如果是对于更新相对较少的页面，可以这样处理，例如新闻、社区通告、或者类似与淘宝网的产品分类信息。但若数据更新频繁，这样做的意义便不大。

网站镜像是个传统的技术，更高级的应用来自流媒体领域的 CDN(Content Delivery Network)，CDN 的概念可以由流媒体数据扩展到图片、视频文件等静态资源的传输。不过，在电子商务领域，很少有这样的应用。

I 原 开源平台的高并发集群思考

目前碰到的高并发应用，需要高性能需求的主要是两个方向

1. 网络
2. 数据库

这两个方面的解决方式其实还是一致的

1. 充分接近单机的性能瓶颈，自我优化
2. 单机搞不定的时候(

数据传输瓶颈:

单位时间内磁盘读写/网络数据包的收发

cpu 计算瓶颈)，把负荷分担给多台机器，就是所谓的负载均衡

网络方面单机的处理

1. 底层包收发处理的模式变化(从 select 模式到 epoll / kevent)
2. 应用模式的变化
 - 2.1 应用层包的构造方式
 - 2.2 应用协议的实现
 - 2.3 包的缓冲模式
 - 2.4 单线程到多线程

网络负载均衡的几个办法

1. 代理模式：代理服务器只管收发包，收到包以后转给后面的应用服务器群（服务器群后可能还会有一堆堆的数据库服务器等等），并且把返回的结果再返回给请求端
2. 虚拟代理 ip: 代理服务器收发包还负载太高，那就增加多台代理服务器，都来管包的转发。这些代理服务器可以用统一的虚拟 ip，也可以单独的 ip
3. p2p: 一些广播的数据可以 p2p 的模式来减轻服务器的网络压力

数据库(指 mysql)单机的处理

1. 数据库本身结构的设计优化（分表，分记录，目的在于保证每个表的记录数在可定的范围内）

2. sql 语句的优化
3. master + slave 模式

数据库集群的处理

1. master + slave 模式 （可有效地处理并发查询）
2. mysql cluster 模式 （可有效地处理并发数据变化）

相关资料:

<http://dev.mysql.com/doc/refman/5.0/en/ndbcluster.html>

I 大型、高负载网站架构和应用初探

时间: 30-45 分钟

开题: 163, sina, sohu 等网站他们有很多应用程序都是 PHP 写的, 为什么他们究竟是如何能做出同时跑几千人甚至上万同时在线应用程序呢?

- 挑选性能更好 web 服务器
 - 单台 Apache web server 性能的极限
 - 选用性能更好的 web server TUX, lighttpd, thttpd ...
 - 动, 静文件分开, 混合使用
- 应用程序优化, Cache 的使用和共享
 - 常见的缓存技术
 - § 生成静态文件
 - § 对象持久化 serialize & unserialize
 - Need for Speed , 在最快的地方做 cache
 - § Linux 系统下的 /dev/shm
 - § tmpfs/ramdisk
 - § php 内置的 shared memory function /IPC
 - § memcached
 - § MySQL 的 HEAP 表
 - 多台主机共享 cache
 - § NFS, memcached, MySQL 优点和缺点比较
- MySQL 数据库优化
 - 配置 my.cnf, 设置更大的 cache size
 - 利用 phpMyAdmin 找出配置瓶颈, 榨干机器的每一点油
 - 集群(热同步, mysql cluster)
- 集群, 提高网站可用性
 - 最简单的集群, 设置多条 A 记录, DNS 轮询, 可用性问题
 - 确保高可用性和伸缩性能的成熟集群解决方案
 - § 通过硬件实现, 如路由器, F5 network
 - § 通过软件或者操作系统实现
 - § 基于内核, 通过修改 TCP/IP 数据报文负载均衡, 并确保伸缩性的 LVS 以及 确保可用性守护进程 ldirectord
 - § 基于 layer 7, 通过 URL 分发的 HAproxy

- 数据共享问题
 - § NFS, Samba, NAS, SAN
 - 案例
- 解决南北互通, 电信和网通速度问题
 - 双线服务器
 - CDN
 - § 根据用户 IP 转换到就近服务器的智能 DNS, dnspod ...
 - § Squid 反向代理, (优点, 缺点)
 - 案例

<http://blog.yening.cn/2007/03/25/226.html#more-226>

I 说说大型高并发高负载网站的系统架构

By Michael

转载请保留出处: 俊麟 Michael's blog (<http://www.toplee.com/blog/?p=71>)

Trackback Url : <http://www.toplee.com/blog/wp-trackback.php?p=71>

我在 CERNET 做过拨号接入平台的搭建, 而后在 Yahoo&3721 从事过搜索引擎前端开发, 又在 MOP 处理过大型社区猫扑大杂烩的架构升级等工作, 同时自己接触和开发过不少大中型网站的模块, 因此在大型网站应对高负载和并发的解决方案上有一些积累和经验, 可以和大家一起探讨一下。

一个小型的网站, 比如个人网站, 可以使用最简单的 html 静态页面就实现了, 配合一些图片达到美化效果, 所有的页面均存放在一个目录下, 这样的网站对系统架构、性能的要求都很简单, 随着互联网业务的不断丰富, 网站相关的技术经过这些年的发展, 已经细分到很细的方方面面, 尤其对于大型网站来说, 所采用的技术更是涉及面非常广, 从硬件到软件、编程语言、数据库、WebServer、防火墙等各个领域都有了很高的要求, 已经不是原来简单的 html 静态网站所能比拟的。

大型网站, 比如门户网站。在面对大量用户访问、高并发请求方面, 基本的解决方案集中在这样几个环节: 使用高性能的服务器、高性能的数据库、高效率的编程语言、还有高性能的 Web 容器。但是除了这几个方面, 还没法根本解决大型网站面临的高负载和高并发问题。

上面提供的几个解决思路在一定程度上也意味着更大的投入, 并且这样的解决思路具备瓶颈, 没有很好的扩展性, 下面我从低成本、高性能和高扩张性的角度来说说我的一些经验。

1、HTML 静态化

其实大家都知道, 效率最高、消耗最小的就是纯静态化的 html 页面, 所以我们尽可能使我们的网站上的页面采用静态页面来实现, 这个最简单的方法其实也是最有效的方法。但是对于大量内容并且频繁更新的网站, 我们无法全部手动去挨个实现, 于是出现了我们常见的信息发布系统 CMS, 像我们常访问的各个门户站点的新闻频道, 甚至他们的其他频道, 都是通过信息发布系统来管理和实现的, 信息发布系统可以实现最简单的信息录入自动生成静态页面, 还能具备频道管理、权限管理、自动抓取等功能, 对于一个大型网站来说, 拥有一套高效、可管理的 CMS 是必不可少的。

除了门户和信息发布类型的网站,对于交互性要求很高的社区类型网站来说,尽可能的静态化也是提高性能的必要手段,将社区内的帖子、文章进行实时的静态化,有更新的时候再重新静态化也是大量使用的策略,像 Mop 的大杂烩就是使用了这样的策略,网易社区等也是如此。目前很多博客也都实现了静态化,我使用的这个 Blog 程序 WordPress 还没有静态化,所以如果面对高负载访问,www.toplee.com 一定不能承受 😊

同时,html 静态化也是某些缓存策略使用的手段,对于系统中频繁使用数据库查询但是内容更新很小的应用,可以考虑使用 html 静态化来实现,比如论坛中论坛的公用设置信息,这些信息目前的主流论坛都可以进行后台管理并且存储再数据库中,这些信息其实大量被前台程序调用,但是更新频率很小,可以考虑将这部分内容进行后台更新的时候进行静态化,这样避免了大量的数据库访问请求。

在进行 html 静态化的时候可以使用一种折中的方法,就是前端使用动态实现,在一定的策略下进行定时静态化和定时判断调用,这个能实现很多灵活性的操作,我开发的台球网站故人居 (www.8zone.cn) 就是使用了这样的方法,我通过设定一些 html 静态化的时间间隔来对动态网站内容进行缓存,达到分担大部分的压力到静态页面上,可以应用于中小型网站的架构上。故人居网站的地址: <http://www.8zone.cn>,顺便提一下,有喜欢台球的朋友多多支持我这个免费网站:)

2、图片服务器分离

大家知道,对于 Web 服务器来说,不管是 Apache、IIS 还是其他容器,图片是最消耗资源的,于是我们有必要将图片与页面进行分离,这是基本上大型网站都会采用的策略,他们都有独立的图片服务器,甚至很多台图片服务器。这样的架构可以降低提供页面访问请求的服务器系统压力,并且可以保证系统不会因为图片问题而崩溃。

在应用服务器和图片服务器上,可以进行不同的配置优化,比如 Apache 在配置 ContentType 的时候可以尽量少支持,尽可能少的 LoadModule,保证更高的系统消耗和执行效率。

我的台球网站故人居 [8zone.cn](http://www.8zone.cn) 也使用了图片服务器架构上的分离,目前是仅仅是架构上分离,物理上没有分离,由于没有钱买更多的服务器:),大家可以看到故人居上的图片连接都是类似 img.9tmd.com 或者 img1.9tmd.com 的 URL。

另外,在处理静态页面或者图片、js 等访问方面,可以考虑使用 [lighttpd](http://lighttpd.org/) 代替 Apache,它提供了更轻量级和更高效的处理能力。

3、数据库集群和库表散列

大型网站都有复杂的应用,这些应用必须使用数据库,那么在面对大量访问的时候,数据库的瓶颈很快就能显现出来,这时一台数据库将很快无法满足应用,于是我们需要使用数据库集群或者库表散列。

在数据库集群方面,很多数据库都有自己的解决方案,Oracle、Sybase 等都有很好的方案,常用的 MySQL 提供的 Master/Slave 也是类似的方案,您使用了什么样的 DB,就参考相应的解决方案来实施即可。

上面提到的数据库集群由于在架构、成本、扩张性方面都会受到所采用 DB 类型的限制,于是我们需要从应用程序的角度来考虑改善系统架构,库表散列是常用并且最有效的解决方案。我们在应用程序中安装业务和应用或者功能模块将数据库进行分离,不同的模块对应不同的数据库或者表,再按照一定的策略对某个页面或者功能进行更小的数据库散列,比如用户表,按照用户 ID 进行表散列,这样就能够低成本的提升系统的性能并且有很好的扩展性。sohu 的论坛就是采用了这样的架构,将论坛的用户、设置、帖子等信息进行数据库分离,然后对帖子、用户按照板

块和 ID 进行散列数据库和表，最终可以在配置文件中进行简单的配置便能让系统随时增加一台低成本数据库进来补充系统性能。

4、缓存

缓存一词搞技术的都接触过，很多地方用到缓存。网站架构和网站开发中的缓存也是非常重要。这里先讲述最基本的两种缓存。高级和分布式的缓存在后面讲述。

架构方面的缓存，对 Apache 比较熟悉的人都能知道 Apache 提供了自己的 mod_proxy 缓存模块，也可以使用外加的 Squid 进行缓存，这两种方式均可以有效的提高 Apache 的访问响应能力。

网站程序开发方面的缓存，Linux 上提供的 Memcached 是常用的缓存方案，不少 web 编程语言都提供 memcache 访问接口，php、perl、c 和 java 都有，可以在 web 开发中使用，可以实时或者 Cron 的把数据、对象等内容进行缓存，策略非常灵活。一些大型社区使用了这样的架构。

另外，在使用 web 语言开发的时候，各种语言基本都有自己的缓存模块和方法，PHP 有 Pear 的 Cache 模块和 eAccelerator 加速和 Cache 模块，还要知名的 Apc、XCache（国人开发的，支持！）php 缓存模块，Java 就更多了，.net 不是很熟悉，相信也肯定有。

5、镜像

镜像是大型网站常采用的提高性能和数据安全性的方式，镜像的技术可以解决不同网络接入商和地域带来的用户访问速度差异，比如 ChinaNet 和 EduNet 之间的差异就促使了很多网站在教育网内搭建镜像站点，数据进行定时更新或者实时更新。在镜像的细节技术方面，这里不阐述太深，有很多专业的现成的解决架构和产品可选。也有廉价的通过软件实现的思路，比如 Linux 上的 rsync 等工具。

6、负载均衡

负载均衡将是大型网站解决高负荷访问和大量并发请求采用的终极解决办法。

负载均衡技术发展了多年，有很多专业的服务提供商和产品可以选择，我个人接触过一些解决方法，其中有两个架构可以给大家做参考。另外有关初级的负载均衡 DNS 轮循和较专业的 CDN 架构就不多说了。

6.1 硬件四层交换

第四层交换使用第三层和第四层信息包的报头信息，根据应用区间识别业务流，将整个区间的业务流分配到合适的应用服务器进行处理。第四层交换功能就象是虚 IP，指向物理服务器。它传输的业务服从的协议多种多样，有 HTTP、FTP、NFS、Telnet 或其他协议。这些业务在物理服务器基础上，需要复杂的载量平衡算法。在 IP 世界，业务类型由终端 TCP 或 UDP 端口地址来决定，在第四层交换中的应用区间则由源端和终端 IP 地址、TCP 和 UDP 端口共同决定。

在硬件四层交换产品领域，有一些知名的产品可以选择，比如 Alteon、F5 等，这些产品很昂贵，但是物有所值，能够提供非常优秀的性能和很灵活的管理能力。Yahoo 中国当初接近 2000 台服务器使用了三四台 Alteon 就搞定了。

6.2 软件四层交换

大家知道了硬件四层交换机的原理后，基于 OSI 模型来实现的软件四层交换也就应运而生，这样的解决方案实现的原理一致，不过性能稍差。但是满足一定量的压力还是游刃有余的，有人说软件实现方式其实更灵活，处理能力完全看你配置的熟悉能力。

软件四层交换我们可以使用 Linux 上常用的 LVS 来解决, LVS 就是 Linux Virtual Server, 他提供了基于心跳线 heartbeat 的实时灾难应对解决方案, 提高系统的鲁棒性, 同时可供了灵活的虚拟 VIP 配置和管理功能, 可以同时满足多种应用需求, 这对于分布式的系统来说必不可少。

一个典型的使用负载均衡的策略就是, 在软件或者硬件四层交换的基础上搭建 squid 集群, 这种思路在很多大型网站包括搜索引擎上被采用, 这样的架构低成本、高性能还有很强的扩张性, 随时往架构里面增减节点都非常容易。这样的架构我准备空了专门详细整理一下和大家探讨。


总结:

对于大型网站来说, 前面提到的每个方法可能都会被同时使用到, Michael 这里介绍得比较浅显, 具体实现过程中很多细节还需要大家慢慢熟悉和体会, 有时一个很小的 squid 参数或者 apache 参数设置, 对于系统性能的影响就会很大, 希望大家一起讨论, 达到抛砖引玉之效。

转载请保留出处: 俊麟 Michael ' s blog (<http://www.toplee.com/blog/?p=71>)

Trackback Url : <http://www.toplee.com/blog/wp-trackback.php?p=71>

This entry is filed under [其他技术](#), [技术交流](#). You can follow any responses to this entry through the [RSS 2.0](#) feed. You can [leave a response](#), or [trackback](#) from your own site.

 (2 votes, average: 6.5 out of 10)

 Loading ...

58 Responses to “说说大型高并发高负载网站的系统架构”

1

pilot says:

April 29th, 2006 at 1:00 pm

[Quote](#)

各模块间或者进程间的通信普遍异步化队列化也相当重要, 可以兼顾轻载重载时的响应性能和系统压力, 数据库压力可以通过 file cache 分解到文件系统, 文件系统 io 压力再通过 mem cache 分解, 效果很不错。

2

Exception says:

April 30th, 2006 at 4:40 pm

[Quote](#)

写得好! 现在, 网上像这样的文章不多, 看完受益匪浅

3

guest says:

May 1st, 2006 at 8:13 am

[Quote](#)

完全胡说八道!

“大家知道, 对于 Web 服务器来说, 不管是 Apache、IIS 还是其他容器, 图片是最消耗资源的”, 你以为是在内存中动态生成图片啊. 无论是什么文件, 在容器输出时只是读文件, 输出给 response 而已, 和是什么文件有什么关系。

关键是静态文件和动态页面之间应该采用不同策略,如静态文件应该尽量缓存,因为无论你请求多少次输出内容都是相同的,如果用户页面中有二十个就没有必要请求二十次,而应该使用缓存.而动态页面每次请求输出都不相同(否则就应该是静态的),所以不应该缓存.

所以即使在同一服务器上也可以对静态和动态资源做不同优化,专门的图片服务器那是为了资源管理的方便,和你说的性能没有关系.

4

Michael says:

May 2nd, 2006 at 1:15 am

Quote

动态的缓存案例估计楼上朋友没有遇到过,在处理 inktomi 的搜索结果的案例中,我们使用的全部是面对动态的缓存,对于同样的关键词和查询条件来说,这样的缓存是非常重要的,对于动态的内容缓存,编程时使用合理的 header 参数可以方便的管理缓存的策略,比如失效时间等。

我们说到有关图片影响性能的问题,一般来说都是出自于我们的大部分访问页面中图片往往比 html 代码占用的流量大,在同等网络带宽的情况下,图片传输需要的时间更长,由于传输需要花很大开销在建立连接上,这会延长用户 client 端与 server 端的 http 连接时长,这对于 apache 来说,并发性能肯定会下降,除非你的返回全部是静态的,那就可以把 httpd.conf 中的 KeepAlives 为 off,这样可以减小连接处理时间,但是如果图片过多会导致建立的连接次数增多,同样消耗性能。

另外我们提到的理论更多的是针对大型集群的案例,在这样的环境下,图片的分离能有效的改进架构,进而影响到性能的提升,要知道我们为什么要谈架构?架构可能为了安全、为了资源分配、也为了更科学的开发和管理,但是终极目都是为了性能。

另外在 RFC1945 的 HTTP 协议文档中很容易找到有关 Mime Type 和 Content length 部分的说明,这样对于理解图片对性能影响是很容易的。

楼上的朋友完全是小人作为,希望别用 guest 跟我忽悠,男人还害怕别人知道你叫啥?再说了,就算说错了也不至于用胡说八道来找茬!大家重在交流和学习,我也不是什么高人,顶多算个普通程序员而已。

5

Ken Kwei says:

June 3rd, 2006 at 3:42 pm

Quote

Michael 您好,这篇文章我看几次了,有一个问题,您的文章中提到了如下一段:

“对于交互性要求很高的社区类型网站来说,尽可能的静态化也是提高性能的必要手段,将社区内的帖子、文章进行实时的静态化,有更新的时候再重新静态化也是大量使用的策略,像 Mop 的大杂烩就是使用了这样的策略,网易社区等也是如此。”

对于大型的站点来说,他的数据库和 Web Server 一般都是分布式的,在多个区域都有部署,当某个地区的用户访问时会对应到一个节点上,如果是对社区内的帖子实时静态化,有更新时再重新静态化,那么在节点之间如何立刻同步呢?数据库端如何实现呢?如果用户看不到的话会以为发帖失败?造成重复发了,那么如何将用户锁定在一个节点上呢,这些怎么解决?谢谢。

6

Michael says:

June 3rd, 2006 at 3:57 pm

[Quote](#)

对于将一个用户锁定在某个节点上是通过四层交换来实现的，一般情况下是这样，如果应用比较小的可以通过程序代码来实现。大型的应用一般通过类似 LVS 和硬件四层交换来管理用户连接，可以制定策略来使用户的连接在生命期内保持在某个节点上。

静态化和同步的策略比较多，一般采用的方法是集中或者分布存储，但是静态化却是通过集中存储来实现的，然后使用前端的 proxy 群来实现缓存和分担压力。

7

javaliker says:

June 10th, 2006 at 6:38 pm

[Quote](#)

希望有空跟你学习请教网站负载问题。

8

barrycmster says:

June 19th, 2006 at 4:14 pm

[Quote](#)

Great website! Bookmarked! I am impressed at your work!

9

heiyeluren says:

June 21st, 2006 at 10:39 am

[Quote](#)

一般对于一个中型网站来说，交互操作非常多，日 PV 百万左右，如何做合理的负载？

10

Michael says:

June 23rd, 2006 at 3:15 pm

[Quote](#)

[heiyeluren on June 21, 2006 at 10:39 am said:](#)

一般对于一个中型网站来说，交互操作非常多，日 PV 百万左右，如何做合理的负载？

交互如果非常多，可以考虑使用集群加 Memory Cache 的方式，把不断变化而且需要同步的数据放入 Memory Cache 里面进行读取，具体的方案还得需要结合具体的情况分析。

11

donald says:

June 27th, 2006 at 5:39 pm

[Quote](#)

请问，如果一个网站处于技术发展期，那么这些优化手段应该先实施哪些后实施哪些呢？或者说从成本（技术、人力和财力成本）方面，哪些先实施能够取得最大效果呢？

12

Michael says:

June 27th, 2006 at 9:16 pm

Quote

[donald on June 27, 2006 at 5:39 pm said:](#)

请问，如果一个网站处于技术发展期，那么这些优化手段应该先实施哪些后实施哪些呢？或者说从成本（技术、人力和财力成本）方面，哪些先实施能够取得最大效果呢？

先从服务器性能优化、代码性能优化方面入手，包括 webserver、dbserver 的优化配置、html 静态化等容易入手的开始，这些环节争取先榨取到最大化的利用率，然后再考虑从架构上增加投入，比如集群、负载均衡等方面，这些都需要在有一定的发展积累之后再考虑比较恰当。

13

donald says:

June 30th, 2006 at 4:39 pm

Quote

恩，多谢 Michael 的耐心讲解

14

Ade says:

July 6th, 2006 at 11:58 am

Quote

写得好,为人也不错.

15

ssbornik says:

July 17th, 2006 at 2:39 pm

Quote

Very good site. Thanks for author!

16

echonow says:

September 1st, 2006 at 2:28 pm

Quote

赞一个先，是一篇很不错的文章，不过要真正掌握里面的东西恐怕还是需要时间和实践！

先问一下关于图片服务器的问题了！

我的台球网站故人居 9tmd.com 也使用了图片服务器架构上的分离，目前是仅仅是架构上分离，物理上没有分离，由于没有钱买更多的服务器:)，大家可以看到故人居上的图片连接都是类似 img.9tmd.com 或者 img1.9tmd.com 的 URL。

这个，楼主这个 img.9tmd.com 是虚拟主机吧，也就是说是一个 apache 提供的服务吧，这样的话对于性能的提高也很有意义吗？还是只是铺垫，为了方便以后的物理分离呢？

17

Michael says:

September 1st, 2006 at 3:05 pm

Quote

[echonow on September 1, 2006 at 2:28 pm said:](#)

赞一个先，是一篇很不错的文章，不过要真正掌握里面的东西恐怕还是需要时间和实践！

先问一下关于图片服务器的问题了！

我的台球网站故人居 9tmd.com 也使用了图片服务器架构上的分离，目前是仅仅是架构上分离，物理上没有分离，由于没有钱买更多的服务器:)，大家可以看到故人居上的图片连接都是类似 img.9tmd.com 或者 img1.9tmd.com 的 URL。

这个，楼主这个 img.9tmd.com 是虚拟主机吧，也就是说是一个 apache 提供的服务吧，这样的话对于性能的提高也很有意义吗？还是只是铺垫，为了方便以后的物理分离呢？

这位朋友说得很对，因为目前只有一台服务器，所以从物理上无法实现真正的分离，暂时使用虚拟主机来实现，是为了程序设计和网站架构上的灵活，如果有了一台新的服务器，我只需要把图片镜像过去或者同步过去，然后把 img.9tmd.com 的 dns 解析到新的服务器上就自然实现了分离，如果现在不从架构和程序上实现，今后这样的分离就会比较痛苦:)

18

echonow says:

September 7th, 2006 at 4:59 pm

Quote

谢谢 lz 的回复，现在主要实现问题是如何能在素材上传时直接传到图片服务器上呢，总不至于每次先传到 web，然后再同步到图片服务器吧

19

Michael says:

September 7th, 2006 at 11:25 pm

Quote

[echonow on September 7, 2006 at 4:59 pm said:](#)

谢谢 lz 的回复，现在主要实现问题是如何能在素材上传时直接传到图片服务器上呢，总不至于每次先传到 web，然后再同步到图片服务器吧

通过 samba 或者 nfs 实现是比较简单的方法。然后使用 squid 缓存来降低访问的负载，提高磁盘性能和延长磁盘使用寿命。

20

echonow says:

September 8th, 2006 at 9:42 am

Quote

多谢楼主的耐心指导，我先研究下，用共享区来存储确实是个不错的想法！

21

Michael says:

September 8th, 2006 at 11:16 am

[Quote](#)

[echonow on September 8, 2006 at 9:42 am said:](#)

多谢楼主的耐心指导，我先研究下，用共享区来存储确实是个不错的想法！

不客气，欢迎常交流！

22

fanstone says:

September 11th, 2006 at 2:26 pm

[Quote](#)

Michael，谢谢你的好文章。仔细看了，包括回复，受益匪浅。

[Michael on June 27, 2006 at 9:16 pm said:](#)

[donald on June 27, 2006 at 5:39 pm said:](#)

请问，如果一个网站处于技术发展期，那么这些优化手段应该先实施哪些后实施哪些呢？

或者说从成本（技术、人力和财力成本）方面，哪些先实施能够取得最大效果呢？

先从服务器性能优化、代码性能优化方面入手，包括 webserver、dbserver 的优化配置、html 静态化等容易入手的开始，这些环节争取先榨取到最大化的利用率，然后再考虑从架构上增加投入，比如集群、负载均衡等方面，这些都需要在有一定的发展积累之后再考虑比较恰当。

尤其这个部分很是有用，因为我也正在建一个电子商务类的网站，由于是前期阶段，费用的问题毕竟有所影响，所以暂且只用了一台服务器囊括过了整个网站。除去前面所说的图片服务器分离，还有什么办法能在网站建设初期尽可能的为后期的发展做好优化（性能优化，系统合理构架，前面说的 webserver、dbserver 优化，后期譬如硬件等扩展尽可能不要过于烦琐等等）？也就是所谓的未雨绸缪了，尽可能在前期考虑到后期如果发展壮大的需求，预先做好系统规划，并且在前期资金不足的情况下尽量做到网站以最优异的性能在运行。关于达到这两个要求，您可以给我一些稍稍详细一点的建议和技术参考吗？谢谢！

看了你的文章，知道你主要关注 *nix 系统架构的，我的是 .net 和 win2003 的，不过我觉得这个影响也不大。主要关注点放在外围的网站优化上。

谢谢！希望能得到您的一些好建议。

23

Michael says:

September 11th, 2006 at 2:55 pm

[Quote](#)

回复 fanstone:

关于如何在网站的前期尽可能低成本的投入，做到性能最大化利用，同时做好后期系统架构的规划，这个问题可以说已经放大到超出技术范畴，不过和技术相关的部分还是有不少需要考虑的。

一个网站的规划关键的就是对阶段性目标的规划，比如预测几个月后达到什么用户级别、存储级别、并发请求数，然后再过几个月又将什么情况，这些预测必须根据具体业务和市场情况进行预估和不断调整的，有了这些预测数据作为参考，就能进行技术架构的规划，否则技术上是无法合理进行架构设计的。

在网站发展规划基础上,考虑今后要提供什么样的应用?有些什么样的域名关系?各个应用之间的业务逻辑和关联是什么?面对什么地域分布的用户提供服务?等等。。。

上面这些问题有助于规划网站服务器和设备投入,同时从技术上可以及早预测到未来将会是一个什么架构,在满足这个架构下的每个节点将需要满足什么条件,就是初期架构的要求。

总的来说,不结合具体业务的技术规划是没有意义的,所以首先是业务规划,也就是产品设计,然后才是技术规划。

24

fanstone says:

September 11th, 2006 at 8:52 pm

[Quote](#)



谢谢解答,我再多看看!

25

Roc says:

March 22nd, 2007 at 11:48 pm

[Quote](#)

很好的文章,楼主说的方法非常适用,目前我们公司的网站也是按照楼主所说的方法进行设计的,效果比较好,利于以后的扩展,另外我再补充一点,其实楼主也说了,网站的域名也需要提前考虑和规划,比如网站的图片内容比较多,可以按应用图片的类型可以根据不同的业务需求采用不同的域名~等,便于日后的扩展和移至,希望楼主能够多发一些这样的好文章。

26

zhang says:

April 3rd, 2007 at 9:08 am

[Quote](#)

图片服务器与主数据分离的问题。

图片是存储在硬盘里好还是存储在数据库里好?

请您分硬盘和数据库两种情况解释下面的疑问。

当存放图片的服务器容量不能满足要求时如何办?

当存放图片的服务器负载不能满足要求时如何办?

谢谢。

27

Michael says:

April 3rd, 2007 at 2:29 pm

[Quote](#)

[zhang on April 3, 2007 at 9:08 am said:](#)

图片服务器与主数据分离的问题。

图片是存储在硬盘里好还是存储在数据库里好?

请您分硬盘和数据库两种情况解释下面的疑问。

当存放图片的服务器容量不能满足要求时如何办?

当存放图片的服务器负载不能满足要求时如何办？

谢谢。

肯定是存储在硬盘里面，出现存储在数据库里面的说法实际上是出自一些虚拟主机或者租用空间的个人网站和企业网站，因为网站数据量小，也为了备份方便，从大型商业网站来说，没有图片存储在数据库里面的大型应用。数据库容量和效率都会是极大的瓶颈。

你提到的后面两个问题。容量和负载基本上是同时要考虑的问题，容量方面，大部分的解决方案都是使用海量存储，比如专业的盘阵，入门级的磁盘柜或者高级的光纤盘阵、局域网盘阵等，这些都是主要的解决方案。记得我原来说过，如果是考虑低成本，一定要自己使用便宜单台服务器来存储，那就需要从程序逻辑上去控制，比如你可以多台同样的服务器来存储，分别提供 NFS 的分区给前端应用使用，在前端应用的程序逻辑中自己去控制存储在哪一台服务器的 NFS 分区上，比如根据 Userid 或者图片 id、或者别的逻辑去进行散列，这个和我们规划大型数据库存储散列分表或者分库存储的逻辑类似。

基本上图片负载高的解决办法有两种，前端 squid 缓存和镜像，通过对存储设备（服务器或者盘阵）使用镜像，可以分布到多台服务器上对外提供图片服务，然后再配合 squid 缓存实现负载的降低和提高用户访问速度。

希望能回答了您的问题。

28

Michael says:

April 3rd, 2007 at 2:41 pm

[Quote](#)

[Roc on March 22, 2007 at 11:48 pm said:](#)

很好的文章，楼主说的方法非常适用，目前我们公司的网站也是按照楼主所说的方法进行设计的，效果比较好，利于以后的扩展，另外我再补充一点，其实楼主也说了，网站的域名也需要提前考虑和规划，比如网站的图片内容比较多，可以按应用图片的类型可以根据不同的业务需求采用不同的域名 img1~imgN 等，便于日后的扩展和移至，希望楼主能够多发一些这样的好文章。

😊欢迎常来交流，还希望能得到你的指点。大家相互学习。

29

zhang says:

April 4th, 2007 at 11:39 pm

[Quote](#)

非常感谢您的回复，
希望将来有合作的机会。

再次感谢。

30

Charles says:

April 9th, 2007 at 2:50 pm

[Quote](#)

刚才一位朋友把你的 BLOG 发给我看，问我是否认识你，所以我就仔细看了一下你的 BLOG，发现这篇文章。

很不错的一篇文章，基本上一个大型网站需要做的事情都已经提及了。我自己也曾任职于三大门户之一，管理超过 100 台的 SQUID 服务器等，希望可以也分享一下我的经验和看法。

1、图片服务器分离

这个观点是我一直以来都非常支持的。特别是如果程序与图片都放在同一个 APACHE 的服务器下，每一个图片的请求都有可能导致一个 HTTPD 进程的调用，而 HTTPD 如果包含有 PHP 模块的时候，就会占用过多的内存，而这个是没有任何必要的。

使用独立的图片服务器不但可以避免以上这个情况，更可以对不同的使用性质的图片设置不同的过期时间，以便同一个用户在不同页面访问相同图片时不会再次从服务器（基于缓存服务器）取数据，不但止快速，而且还省了带宽。还有就是，对于缓存的时间上，亦可以做调立的调节。

在我过往所管理的图片服务器中，不但止是将图片与应用及页面中分离出来，还是为不同性质的图片启用不同的域名。以缓解不同性质图片带来的压力。例如 photo.img.domain.com 这个域名是为了摄影服务的，平时使用 5 台 CACHE，但到了 5.1 长假期后，就有可能需要独立为他增加至 10 台。而增加的这 5 台可以从其他负载较低的图片服务器中调动过来临时使用。

2、数据库集群

一套 ORACLE RAC 的集群布置大概在 40W 左右，这个价格对于一般公司来说，是没有必要的。因为 WEB 的应用逻辑相对较简单，而 ORACLE 这些大型数据库的价值在于数据挖掘，而不在于简单的存储。所以选择 MySQL 或 PostgreSQL 会实际一些。

简单的 MySQL 复制就可以实现较好的效果。读的时候从 SLAVE 读，写的时候才到 MASTER 上更新。实际的情况下，MySQL 的复制性能非常好，基本上不会带来太高的更新延时。使用 balance (<http://www.inlab.de/balance.html>) 这个软件，在本地 (127.0.0.1) 监听 3306 端口，再映射多个 SLAVE 数据库，可以实现读取的负载均衡。

3、图片保存于磁盘还是数据库？

对于这个问题，我亦有认真地考虑过。如果是在 ext3 的文件系统下，建 3W 个目录就到极限了，而使用 xfs 的话就没有这个限制。图片的存储，如果需要是大量的保存，必须要分隔成很多个小目录，否则就会有 ext3 只能建 3W 目录的限制，而且文件数及目录数太多会影响磁盘性能。还没有算上空间占用浪费等问题。

更更重要的是，对于一个大量小文件的数据备份，要占用极大的资源和非常长的时间。在这些问题上，可能将图片保存在数据库是个另外的选择。

可以尝试将图片保存到数据库，前端用 PHP 程序返回实际的图片，再在前端放置一个 SQUID 的服务器，可以避免性能问题。那么图片的备份问题，亦可以利用 MySQL 的数据复制机制来实现。这个问题就可以得到较好的解决了。

4、页面的静态化我就不说了，我自己做的 wordpress 就完全实现了静态化，同时能很好地兼顾动态数据的生成。

5、缓存

我自己之前也提出过使用 memcached，但实际使用中不是非常特别的理想。当然，各个应用环境不一致会有不一致的使用结果，这个并不重要。只要自己觉得好用就用。

6、软件四层交换

LVS 的性能非常好，我有朋友的网站使用了 LVS 来做负责均衡的调度器，数据量非常大都可以轻松支撑。当然是使用了 DR 的方式。

其实我自己还想过可以用 LVS 来做 CDN 的调度。例如北京的 BGP 机房接受用户的请求，然后通过 LVS 的 TUN 方式，将请求调度到电信或网通机房的实际物理服务器上，直接向用户返回数据。

这种是 WAN 的调度，F5 这些硬件设备也应用这样的技术。不过使用 LVS 来实现费用就大大降低。

以上都只属个人观点，能力有限，希望对大家有帮助。：)

31

Michael says:

April 9th, 2007 at 8:17 pm

Quote

很少见到有朋友能在我得 blog 上留下这么多有价值的东西，代表别的可能看到这篇文章的朋友一起感谢你。

balance (<http://www.inlab.de/balance.html>) 这个东西准备看一下。

32

Michael says:

April 16th, 2007 at 1:29 am

Quote

如果说 3Par 的光纤存储局域网技术细节，我无法给您太多解释，我对他们的产品没有接触也没有了解，不过从 SAN 的概念上是可以知道大概框架的，它也是一种基于光纤通道的存储局域网，可以支持远距离传输和较高的系统扩展性，传统的 SAN 使用专门的 FC 光通道 SCSI 磁盘阵列，从你提供的内容来看，3Par 这个东西建立在低成本的 SATA 或 FATA 磁盘阵列基础上，这一方面能降低成本，同时估计 3Par 在技术上有创新和改进，从而提供了廉价的高性能存储应用。

这个东西细节只有他们自己知道，你就知道这是个商业的 SAN（存储局域网，说白了也是盘阵，只是通过光纤通道独立于系统外的）。

33

zhang says:

April 16th, 2007 at 2:10 am

Quote

myspace 和美国的许多银行都更换为了 3Par

请您在百忙之中核实一下，是否确实像说的那么好。

下面是摘抄：

Priceline.com 是一家以销售空座机票为主的网络公司，客户数量多达 680 万。该公司近期正在内部实施一项大规模的 SAN 系统整合计划，一口气购进了 5 套 3PARdata 的服务器系统，用以替代现有的上百台 Sun 存储阵列。如果该方案部署成功的话，将有望为 Priceline.com 节省大量的存储管理时间、资本开销及系统维护费用。

Priceline.com 之前一直在使用的 SAN 系统是由 50 台光纤磁盘阵列、50 台 SCSI 磁盘阵列和

15 台存储服务器构成的。此次，该公司一举购入了 5 台 3Par S400 InServ Storage Servers 存储服务器，用以替代原来的服务器系统，使得设备整体能耗、占用空间及散热一举降低了 60%。整个系统部署下来，总存储容量将逼近 30TB。

Priceline 的首席信息官 Ron Rose 拒绝透露该公司之前所使用的 SAN 系统设备的供应商名称，不过，消息灵通人士表示，PriceLine 原来的存储环境是由不同型号的 Sun 系统混合搭建而成的。

“我们并不愿意随便更换系统供应商，不过，3Par 的存储系统所具备的高投资回报率，实在令人难以抗拒，” Rose 介绍说，“我们给了原来的设备供应商以足够的适应时间，希望它们的存储设备也能够提供与 3Par 一样的效能，最后，我们失望了。如果换成 3Par 的存储系统的话，短期内就可以立刻见到成效。”

Rose 接着补充说，“原先使用的那套 SAN 系统，并没有太多让我们不满意的地方，除了欠缺一点灵活性之外：系统的配置方案堪称不错，但并不是最优化的。使用了大量价格偏贵的光纤磁盘，许多 SAN 端口都是闲置的。”

自从更换成 3Par 的磁盘阵列之后，该公司存储系统的端口数量从 90 个骤减为 24 个。“我们购买的软件许可证都是按端口数量来收费的。每增加一个端口，需要额外支付 500-1,500 美元，单单这一项，就为我们节省了一笔相当可观的开支，” Rose 解释说。而且，一旦启用 3Par 的精简自动配置软件，系统资源利用率将有望提升 30%，至少在近一段时间内该公司不必考虑添置新的磁盘系统。

精简自动配置技术最大的功效就在于它能够按照应用程序的实际需求来分配存储资源，有效地降低了空间闲置率。如果当前运行的应用程序需要额外的存储资源的话，该软件将在不干扰应用程序正常运行的前提下，基于“按需”和“公用”的原则来自动发放资源空间，避免了人力干预，至少为存储管理员减轻了一半以上的工作量。

3Par 的磁盘阵列是由低成本的 SATA 和 FATA（即：低成本光纤信道接口）磁盘驱动器构成的，而并非昂贵的高效能 FC 磁盘，大大降低了系统整体成本。

3Par 推出的 SAN 解决方案，实际上是遵循了“允许多个分布式介质服务器共享通过光纤信道 SAN 连接的公共的集中化存储设备”的设计理念。“这样一来，就不必给所有的存储设备都外挂一个代理服务程序了，” Rose 介绍说。出于容灾容错和负载均衡的考虑，Priceline 搭建了两个生产站点，每一个站点都部署了一套 3Par SAN 系统。此外，Priceline 还购买了两台 3Par Inservs 服务器，安置在主数据中心内，专门用于存放镜像文件。第 5 台服务器设置在 Priceline 的企业资料处理中心内，用于存放数据仓库；第 6 台服务器设置在实验室内，专门用于进行实际网站压力测试。

MySpace 目前采用了一种新型 SAN 设备——来自加利福尼亚州弗里蒙特的 3PARdata。在 3PAR 的系统里，仍能在逻辑上按容量划分数据存储，但它不再被绑定到特定磁盘或磁盘簇，而是散布于大量磁盘。这就使均分数据访问负荷成为可能。当数据库需要写入一组数据时，任何空闲磁盘都可以马上完成这项工作，而不再像以前那样阻塞在可能已经过载的磁盘阵列处。而且，因为多个磁盘都有数据副本，读取数据时，也不会使 SAN 的任何组件过载。

3PAR 宣布，VoIP 服务供应商 Cbeyond Communications 已向它订购了两套 InServ 存储服务器，一套应用于该公司的可操作支持系统，一套应用于测试和开发系统环境。3PAR 的总部设在亚特兰大，该公司的产品多销往美国各州首府和大城市，比如说亚特兰大、达拉斯、丹佛、休斯顿、芝加哥，等等。截至目前为止，3PAR 售出的服务器数量已超过了 15,000 台，许多客户都是来自于各行各业的龙头企业，它们之所以挑选 3PAR 的产品，主要是看中了它所具备的高性能、可扩展性、操作简单、无比伦比的性价比等优点，另外，3PAR 推出的服务器系列具有高度的集成性

能，适合应用于高速度的 T1 互联网接入、本地和长途语音服务、虚拟主机（Web hosting）、电子邮件、电话会议和虚拟个人网络（VPN）等服务领域。

亿万用户网站 MySpace 的成功秘密

◎ 文 / David F. Carr 译 / 罗小平

高速增长的访问量给社区网络的技术体系带来了巨大挑战。MySpace 的开发者的多年来不断重构站点软件、数据库和存储系统，以期与自身的成长同步——目前，该站点月访问量已达 400 亿。绝大多数网站需要应对的流量都不及 MySpace 的一小部分，但那些指望迈入庞大在线市场的人，可以从 MySpace 的成长过程学到知识。

MySpace 开发人员已经多次重构站点软件、数据库和存储系统，以满足爆炸性的成长需要，但此工作永不会停息。“就像粉刷金门大桥，工作完成之时，就是重新来过之日。”（译者注：意指工人从桥头开始粉刷，当到达桥尾时，桥头涂料已经剥落，必须重新开始）MySpace 技术副总裁 Jim Benedetto 说。

既然如此，MySpace 的技术还有何可学之处？因为 MySpace 事实上已经解决了很多系统扩展性问题，才能走到今天。

Benedetto 说他的项目组有很多教训必须总结，他们仍在学习，路漫漫而修远。他们当前需要改进的工作包括实现更灵活的数据缓存系统，以及为避免再次出现类似 7 月瘫痪事件的地理上分布式架构。

背景知识

当然，这么多的用户不停发布消息、撰写评论或者更新个人资料，甚至一些人整天都泡在 Space 上，必然给 MySpace 的技术工作带来前所未有的挑战。而传统新闻站点的绝大多数内容都是由编辑团队整理后主动提供给用户消费，它们的内容数据库通常可以优化为只读模式，因为用户评论等引起的增加和更新操作很少。而 MySpace 是由用户提供内容，数据库很大比例的操作都是插入和更新，而非读取。

浏览 MySpace 上的任何个人资料时，系统都必须先查询数据库，然后动态创建页面。当然，通过数据缓存，可以减轻数据库的压力，但这种方案必须解决原始数据被用户频繁更新带来的同步问题。

MySpace 的站点架构已经历了 5 个版本——每次都是用户数达到一个里程碑后，必须做大量的调整和优化。Benedetto 说，“但我们始终跟不上形势的发展速度。我们重构重构再重构，一步步挪到今天”。

在每个里程碑，站点负担都会超过底层系统部分组件的最大载荷，特别是数据库和存储系统。接着，功能出现问题，用户失声尖叫。最后，技术团队必须为此修订系统策略。

虽然自 2005 年早期，站点账户数超过 7 百万后，系统架构到目前为止保持了相对稳定，但 MySpace 仍然在为 SQL Server 支持的同时连接数等方面继续攻坚，Benedetto 说，“我们已经尽可能把事情做到最好”。

里程碑一：50 万账户

按 Benedetto 的说法，MySpace 最初的系统很小，只有两台 Web 服务器和一个数据库服务器。那时使用的是 Dell 双 CPU、4G 内存的系统。

单个数据库就意味着所有数据都存储在一个地方，再由两台 Web 服务器分担处理用户请求的工作量。但就像 MySpace 后来的几次底层系统修订时的情况一样，三服务器架构很快不堪重负。此后一个时期内，MySpace 基本是通过添置更多 Web 服务器来对付用户暴增问题的。

但到在 2004 年早期，MySpace 用户数增长到 50 万后，数据库服务器也已开始汗流浹背。但和 Web 服务器不同，增加数据库可没那么简单。如果一个站点由多个数据库支持，设计者必须考虑的是，如何在保证数据一致性的前提下，让多个数据库分担压力。在第二代架构中，MySpace 运行在 3 个 SQL Server 数据库服务器上——一个为主，所有的新数据都向它提交，然后由它复制到其他两个；另两个全力向用户供给数据，用以在博客和个人资料栏显示。这种方式在一段时间内效果很好——只要增加数据库服务器，加大硬盘，就可以应对用户数和访问量的增加。

里程碑二：1-2 百万账户

MySpace 注册数到达 1 百万至 2 百万区间后，数据库服务器开始受制于 I/O 容量——即它们存取数据的速度。而当时才是 2004 年中，距离上次数据库系统调整不过数月。用户的提交请求被阻塞，就像千人乐迷要挤进只能容纳几百人的夜总会，站点开始遭遇“主要矛盾”，Benedetto 说，这意味着 MySpace 永远都会轻度落后于用户需求。

“有人花 5 分钟都无法完成留言，因此用户总是抱怨说网站已经完蛋了。”他补充道。这一次的数据库架构按照垂直分割模式设计，不同的数据库服务于站点的不同功能，如登录、用户资料和博客。于是，站点的扩展性问题看似又可以告一段落了，可以歇一阵子。垂直分割策略利于多个数据库分担访问压力，当用户要求增加新功能时，MySpace 将投入新的数据库予以支持它。账户到达 2 百万后，MySpace 还从存储设备与数据库服务器直接交互的方式切换到 SAN（Storage Area Network，存储区域网络）——用高带宽、专门设计的网络将大量磁盘存储设备连接在一起，而数据库连接到 SAN。这项措施极大提升了系统性能、正常运行时间和可靠性，Benedetto 说。

里程碑三：3 百万账户

当用户继续增加到 3 百万后，垂直分割策略也开始难以为继。尽管站点的各个应用被设计得高度独立，但有些信息必须共享。在这个架构里，每个数据库必须有各自的用户表副本——MySpace 授权用户的电子花名册。这就意味着一个用户注册时，该条账户记录必须在 9 个不同数据库上分别创建。但在个别情况下，如果其中某台数据库服务器临时不可到达，对应事务就会失败，从而造成账户非完全创建，最终导致此用户的该项服务无效。

另外一个问题是，个别应用如博客增长太快，那么专门为它服务的数据库就有巨大压力。2004 年中，MySpace 面临 Web 开发者称之为“向上扩展”对“向外扩展”（译者注：Scale Up 和 Scale Out，也称硬件扩展和软件扩展）的抉择——要么扩展到更大更强、也更昂贵的服务器上，要么部署大量相对便宜的服务器来分担数据库压力。一般来说，大型站点倾向于向外扩展，因为这将让它们得以保留通过增加服务器以提升系统能力的后路。

但成功地向外扩展架构必须解决复杂的分布式计算问题，大型站点如 Google、Yahoo 和 Amazon.com，都必须自行研发大量相关技术。以 Google 为例，它构建了自己的分布式文件系统。另外，向外扩展策略还需要大量重写原来软件，以保证系统能在分布式服务器上运行。“搞不好，开发人员的所有工作都将白费”，Benedetto 说。

因此，MySpace 首先将重点放在了向上扩展上，花费了大约 1 个半月时间研究升级到 32CPU 服务器以管理更大数据库的问题。Benedetto 说，“那时候，这个方案看似可能解决一切问题。”如稳定性，更棒的是对现有软件几乎没有改动要求。

糟糕的是，高端服务器极其昂贵，是购置同样处理能力和内存速度的多台服务器总和的很多倍。而且，站点架构师预测，从长期来看，即便是巨型数据库，最后也会不堪重负，Benedetto 说，“换句话讲，只要增长趋势存在，我们最后无论如何都要走上向外扩展的道路。”

因此，MySpace 最终将目光移到分布式计算架构——它在物理上分布的众多服务器，整体必须逻辑上等同于单台机器。拿数据库来说，就不能再像过去那样将应用拆分，再以不同数据库分别支

持，而必须将整个站点看作一个应用。现在，数据库模型里只有一个用户表，支持博客、个人资料和其他核心功能的数据都存储在相同数据库。

既然所有的核心数据逻辑上都组织到一个数据库，那么 MySpace 必须找到新的办法以分担负荷——显然，运行在普通硬件上的单个数据库服务器是无能为力的。这次，不再按站点功能和应用分割数据库，MySpace 开始将它的用户按每百万一组分割，然后将各组的全部数据分别存入独立的 SQL Server 实例。目前，MySpace 的每台数据库服务器实际运行两个 SQL Server 实例，也就是说每台服务器服务大约 2 百万用户。Benedetto 指出，以后还可以按照这种模式以更小粒度划分架构，从而优化负荷分担。

当然，还是有一个特殊数据库保存了所有账户的名称和密码。用户登录后，保存了他们其他数据的数据库再接管服务。特殊数据库的用户表虽然庞大，但它只负责用户登录，功能单一，所以负荷还是比较容易控制的。

里程碑四：9 百万到 1 千 7 百万账户

2005 年早期，账户达到 9 百万后，MySpace 开始用 Microsoft 的 C# 编写 ASP.NET 程序。C# 是 C 语言的最新派生语言，吸收了 C++ 和 Java 的优点，依托于 Microsoft .NET 框架（Microsoft 为软件组件化和分布式计算而设计的模型架构）。ASP.NET 则由编写 Web 站点脚本的 ASP 技术演化而来，是 Microsoft 目前主推的 Web 站点编程环境。

可以说是立竿见影，MySpace 马上就发现 ASP.NET 程序运行更有效率，与 ColdFusion 相比，完成同样任务需消耗的处理能力更小。据技术总监 Whittcomb 说，新代码需要 150 台服务器完成的工作，如果用 ColdFusion 则需要 246 台。Benedetto 还指出，性能上升的另一个原因可能是在变换软件平台，并用新语言重写代码的过程中，程序员复审并优化了一些功能流程。

最终，MySpace 开始大规模迁移到 ASP.NET。即便剩余的少部分 ColdFusion 代码，也从 ColdFusion 服务器搬到了 ASP.NET，因为他们得到了 BlueDragon.NET（乔治亚州阿尔法利塔 New Atlanta Communications 公司的产品，它能将 ColdFusion 代码自动重新编译到 Microsoft 平台）的帮助。

账户达到 1 千万时，MySpace 再次遭遇存储瓶颈问题。SAN 的引入解决了早期一些性能问题，但站点目前的要求已经开始周期性超越 SAN 的 I/O 容量——即它从磁盘存储系统读写数据的极限速度。

原因之一是每数据库 1 百万账户的分割策略，通常情况下的确可以将压力均分到各台服务器，但现实并非一成不变。比如第七台账户数据库上线后，仅仅 7 天就被塞满了，主要原因是佛罗里达一个乐队的歌迷疯狂注册。

某个数据库可能因为任何原因，在任何时候遭遇主要负荷，这时，SAN 中绑定到该数据库的磁盘存储设备簇就可能过载。“SAN 让磁盘 I/O 能力大幅提升了，但将它们绑定到特定数据库的做法是错误的。” Benedetto 说。

最初，MySpace 通过定期重新分配 SAN 中数据，以让其更为均衡的方法基本解决了这个问题，但这是一个人工过程，“大概需要两个人全职工作。” Benedetto 说。

长期解决方案是迁移到虚拟存储体系上，这样，整个 SAN 被当作一个巨型存储池，不再要求每个磁盘为特定应用服务。MySpace 目前采用了一种新型 SAN 设备——来自加利福尼亚州弗里蒙特的 3PARdata。

在 3PAR 的系统里，仍能在逻辑上按容量划分数据存储，但它不再被绑定到特定磁盘或磁盘簇，而是散布于大量磁盘。这就使均分数据访问负荷成为可能。当数据库需要写入一组数据时，任何空闲磁盘都可以马上完成这项工作，而不再像以前那样阻塞在可能已经过载的磁盘阵列处。而且，因为多个磁盘都有数据副本，读取数据时，也不会使 SAN 的任何组件过载。

当 2005 年春天账户数达到 1 千 7 百万时，MySpace 又启用了新的策略以减轻存储系统压力，即

增加数据缓存层——位于 Web 服务器和数据库服务器之间，其唯一职能是在内存中建立被频繁请求数据对象的副本，如此一来，不访问数据库也可以向 Web 应用供给数据。换句话说，100 个用户请求同一份资料，以前需要查询数据库 100 次，而现在只需 1 次，其余都可从缓存数据中获得。当然如果页面变化，缓存的数据必须从内存擦除，然后重新从数据库获取——但在此之前，数据库的压力已经大大减轻，整个站点的性能得到提升。

缓存区还为那些不需要记入数据库的数据提供了驿站，比如为跟踪用户会话而创建的临时文件——Benedetto 坦言他需要在这方面补课，“我是数据库存储狂热分子，因此我总是想着将万事万物都存到数据库。”但将会话跟踪这类的数据也存到数据库，站点将陷入泥沼。

增加缓存服务器是“一开始就应该做的事情，但我们成长太快，以致于没有时间坐下来好好研究这件事情。”Benedetto 补充道。

里程碑五：2 千 6 百万账户

2005 年中期，服务账户数达到 2 千 6 百万时，MySpace 切换到了还处于 beta 测试的 SQL Server 2005。转换何太急？主流看法是 2005 版支持 64 位处理器。但 Benedetto 说，“这不是主要原因，尽管这也很重要；主要还是因为我们对内存的渴求。”支持 64 位的数据库可以管理更多内存。更多内存就意味着更高的性能和更大的容量。原来运行 32 位版本的 SQL Server 服务器，能同时使用的内存最多只有 4G。切换到 64 位，就好像加粗了输水管的直径。升级到 SQL Server 2005 和 64 位 Windows Server 2003 后，MySpace 每台服务器配备了 32G 内存，后于 2006 年再次将配置标准提升到 64G。

意外错误

如果没有对系统架构的历次修改与升级，MySpace 根本不可能走到今天。但是，为什么系统还经常吃撑着了？很多用户抱怨的“意外错误”是怎么引起的呢？

原因之一是 MySpace 对 Microsoft 的 Web 技术的应用已经进入连 Microsoft 自己也才刚刚开始探索的领域。比如 11 月，超出 SQL Server 最大同时连接数，MySpace 系统崩溃。Benedetto 说，这类可能引发系统崩溃的情况大概三天才会出现一次，但仍然过于频繁了，以致惹人恼怒。一旦数据库罢工，“无论这种情况什么时候发生，未缓存的数据都不能从 SQL Server 获得，那么你就必然看到一个‘意外错误’提示。”他解释说。

去年夏天，MySpace 的 Windows 2003 多次自动停止服务。后来发现是操作系统一个内置功能惹的祸——预防分布式拒绝服务攻击（黑客使用很多客户机向服务器发起大量连接请求，以致服务器瘫痪）。MySpace 和其他很多顶级大站点一样，肯定会经常遭受攻击，但它应该从网络级而不是依靠 Windows 本身的功能来解决问题——否则，大量 MySpace 合法用户连接时也会引起服务器反击。

“我们花了大约一个月时间寻找 Windows 2003 服务器自动停止的原因。”Benedetto 说。最后，通过 Microsoft 的帮助，他们才知道该怎么通知服务器：“别开枪，是友军。”

紧接着是在去年 7 月某个周日晚上，MySpace 总部所在地洛杉矶停电，造成整个系统停运 12 小时。大型 Web 站点通常要在地理上分布配置多个数据中心以预防单点故障。本来，MySpace 还有其他两个数据中心以应对突发事件，但 Web 服务器都依赖于部署在洛杉矶的 SAN。没有洛杉矶的 SAN，Web 服务器除了恳求你耐心等待，不能提供任何服务。

Benedetto 说，主数据中心的可靠性通过下列措施保证：可接入两张不同电网，另有后备电源和一台储备有 30 天燃料的发电机。但在这次事故中，不仅两张电网失效，而且在切换到备份电源的过程中，操作员烧掉了主动力线路。

2007 年中，MySpace 在另两个后备站点上也建设了 SAN。这对分担负荷大有帮助——正常情况下，每个 SAN 都能负担三分之一的数据访问量。而在紧急情况下，任何一个站点都可以独立支撑整个服务，Benedetto 说。

MySpace 仍然在为提高稳定性奋斗，虽然很多用户表示了足够信任且能原谅偶现的错误页面。

“作为开发人员，我憎恶 Bug，它太气人了。” Dan Tanner 这个 31 岁的德克萨斯软件工程师说，他通过 MySpace 重新联系到了高中和大学同学。“不过，MySpace 对我们的用处很大，因此我们可以原谅偶发的故障和错误。” Tanner 说，如果站点某天出现故障甚至崩溃，恢复以后他还是会继续使用。

这就是为什么 Drew 在论坛里咆哮时，大部分用户都告诉他应该保持平静，如果等几分钟，问题就会解决的原因。Drew 无法平静，他写道，“我已经两次给 MySpace 发邮件，而它说一小时前还是正常的，现在出了点问题……完全是一堆废话。”另一个用户回复说，“毕竟它是免费的。” Benedetto 坦承 100% 的可靠性不是他的目标。“它不是银行，而是一个免费的服务。”他说。

换句话说，MySpace 的偶发故障可能造成某人最后更新的个人资料丢失，但并不意味着网站弄丢了用户的钱财。“关键是要认识到，与保证站点性能相比，丢失少许数据的故障是可接受的。” Benedetto 说。所以，MySpace 甘冒丢失 2 分钟到 2 小时内任意点数据的危险，在 SQL Server 配置里延长了“checkpoint”操作——它将待更新数据永久记录到磁盘——的间隔时间，因为这样做可以加快数据库的运行。

Benedetto 说，同样，开发人员还经常在几个小时内就完成构思、编码、测试和发布全过程。这有引入 Bug 的风险，但这样做可以更快实现新功能。而且，因为进行大规模真实测试不具可行性，他们的测试通常是在仅以部分活跃用户为对象，且用户对软件新功能和改进不知就里的情况下进行的。因为事实上不可能做真实的加载测试，他们做的测试通常都是针对站点。

“我们犯过大量错误，” Benedetto 说，“但到头来，我认为我们做对的还是比做错的多。”

34

zhang says:

April 16th, 2007 at 2:15 am

[Quote](#)

了解联合数据库服务器

为达到最大型网站所需的高性能级别，多层系统一般在多个服务器之间平衡每一层的处理负荷。SQL Server 2005 通过对 SQL Server 数据库中的数据进行水平分区，在一组服务器之间分摊数据库处理负荷。这些服务器独立管理，但协作处理应用程序的数据库请求；这样一组协作服务器称为“联合体”。

只有在应用程序将每个 SQL 语句发送到包含该语句所需的大部分数据的成员服务器时，联合数据库层才能达到非常高的性能级别。这称为使用语句所需的数据来配置 SQL 语句。使用所需的数据来配置 SQL 语句不是联合服务器所特有的要求。群集系统也有此要求。

虽然服务器联合体与单个数据库服务器对应用程序来说是一样的，但在实现数据库服务层的方式上存在内部差异，

35

Michael says:

April 16th, 2007 at 3:18 am

[Quote](#)

关于 MySpace 是否使用了 3Par 的 SAN，并且起到多大的关键作用，我也无法考证，也许可以通过在 MySpace 工作的朋友可以了解到，但是从各种数据和一些案例来看，3Par 的确可以改善成本过高和存储 I/O 性能问题，但是实际应用中，除非电信、银行或者真的类似 MySpace 这样规模的站点，的确很少遇到存储连 SAN 都无法满足的情况，另外，对于数据库方面，据我知道，凡电

信、金融和互联网上电子商务关键数据应用，基本上 Oracle 才是最终的解决方案。包括我曾经工作的 Yahoo，他们全球超过 70% 以上应用使用 MySQL，但是和钱相关的或者丢失数据会承担责任的应用，都是使用 Oracle。在 UDB 方面，我相信 Yahoo 的用户数一定超过 MySpace 的几千万。

事实上，国内最值得研究的案例应该是腾讯，腾讯目前的数据量一定是惊人的，在和周小旻的一次短暂对话中知道腾讯的架构专家自己实现了大部分的技术，细节我无法得知。

36

Michael says:

April 16th, 2007 at 3:23 am

[Quote](#)

图片存储到数据库我依然不赞同，不过一定要这么做也不是不可以，您提到的使用 CGI 程序输出到用户客户端，基本上每种 web 编程语言都支持，只要能输出标准的 HTTP Header 信息就可以了，比如 PHP 可以使用 `header(" content-type: image/jpeg\r\n");` 语句输出当前 http 返回的文件 mime 类型为图片，同时还有更多的 `header()` 函数可以输出的 HTTP Header 信息，包括 `content-length` 之类的（提供 range 断点续传需要），具体的可以参考 PHP 的手册。另外，perl、asp、jsp 这些都提供类似的实现方法，这不是语言问题，而是一个 HTTP 协议问题。

37

zhang says:

April 16th, 2007 at 8:52 am

[Quote](#)

早晨，其实已经是上午，起床后，
看到您凌晨 3: 23 的回复，非常感动。
一定注意身体。
好像您还没有太太，
太太和孩子也像正规程序一样，会良好地调节您的身体。
千万不要使用野程序调节身体，会中毒。

开个玩笑。

38

zhang says:

April 16th, 2007 at 8:59 am

[Quote](#)

看到您凌晨 3: 23 的回复，
非常感动！
一定注意身体，
好像您还没有太太，
太太和孩子就像正规程序一样，能够良好地调节您的身体，
千万不要使用野程序调节身体，会中毒。

开个玩笑。

39

Michael says:

April 16th, 2007 at 11:04 am

Quote

[zhang on April 16, 2007 at 8:59 am said:](#)

看到您凌晨 3: 23 的回复，
非常感动！
一定注意身体，
好像您还没有太太，
太太和孩子就像正规程序一样，能够良好地调节您的身体，
千万不要使用野程序调节身体，会中毒。

开个玩笑。

哈哈，最近我是有点疯狂，不过从大学开始，似乎就习惯了晚睡，我基本多年都保持 2 点左右睡觉，8 点左右起床，昨晚有点夸张，因为看一个文档和写一些东西一口气就到 3 点多了，临睡前看到您的留言，顺便就回复了。

40

myld says:

April 18th, 2007 at 1:38 pm

Quote

感谢楼主写了这么好的文章，谢谢！！

41

楓之谷外掛 says:

April 27th, 2007 at 11:04 pm

Quote

看了你的文章，很有感覺的說。我自己也做網站，希望可以多多交流一下，大家保持聯繫。

<http://www.gameon9.com/>

<http://www.gameon9.com.tw/>

42

南半球 says:

May 9th, 2007 at 8:22 pm

Quote

关于两位老大讨论的：图片保存于磁盘还是数据库

个人觉得数据库存文件的话，查询速度可能快点，但数据量很大的时候要加上索引，这样添加记录的速度就慢了

mysql 对大数据量的处理能力还不是很强，上千万条记录时，性能也会下降

数据库另外一个瓶颈问题就是连接

用数据库，就要调用后台程序（JSP/JAVA, PHP 等）连接数据库，而数据库的连接连接、传输数据都要耗费系统资源。数据库的连接数也是个瓶颈问题。曾经写过一个很烂的程序，每秒访问 3 到 5 次的数据库，结果一天下来要连接 20 多万次数据库，把对方的 mysql 数据库搞瘫痪了。

43

zhang says:

May 19th, 2007 at 12:07 am

[Quote](#)

抽空儿回这里浏览了一下，
有收获，
“写真照”换了，显得更帅了。

ok

44

Michael says:

May 19th, 2007 at 12:17 am

[Quote](#)

[zhang on May 19, 2007 at 12:07 am said:](#)

抽空儿回这里浏览了一下，
有收获，
“写真照”换了，显得更帅了。

ok

哈哈，让您见笑了 😊

45

David says:

May 30th, 2007 at 3:27 pm

[Quote](#)

很好，虽然我不是做 web 的，但看了还是收益良多。

46

pig345 says:

June 13th, 2007 at 10:23 am

[Quote](#)

感谢 Mi chael

47

疯子日记 says:

June 13th, 2007 at 10:12 pm

[Quote](#)

回复: 说说大型高并发高负载网站的系统架构 ...

好文章! 学习中.....

48

terry says:

June 15th, 2007 at 4:29 pm

[Quote](#)

推荐 ngi nx

49

7u5 says:

June 16th, 2007 at 11:54 pm

[Quote](#)

拜读

50

Michael says:

June 16th, 2007 at 11:59 pm

[Quote](#)

[terry on June 15, 2007 at 4:29 pm said:](#)

推荐 ngi nx

欢迎分享 Ngi nx 方面的经验:)

51

说说大型高并发高负载网站的系统架构 - 红色的河 says:

June 21st, 2007 at 11:40 pm

[Quote](#)

[...] 来源: <http://www.toplee.com/blog/archives/71.html> 时间: 11:40 下午 | 分类: 技术文摘 标签: 系统架构, 大型网站, 性能优化 [...]

52

laoyao2k says:

June 23rd, 2007 at 11:35 am

[Quote](#)

看到大家都推荐图片分离,我也知道这样很好,但页面里的图片的绝对网址是开发的时候就写进去的,还是最终执行的时候替换的呢?

如果是开发原始网页就写进去的,那本地调试的时候又是怎么显示的呢?

如果是最终执行的时候替换的话,是用的什么方法呢?

53

Michael says:

June 23rd, 2007 at 8:21 pm

[Quote](#)

都可以,写到配置文件里面就可以,或者用全局变量定义,方法很多也都能实现,哪怕写死了在开发的时候把本地调试也都配好图片 server,在 hosts 文件里面指定一下 ip 到本地就可以了。

假设用最终执行时候的替换,就配置你的 apache 或者别的 server 上的 mod_rewrite 模块来实现,具体的参照相关文档。

54

laoyao2k says:

June 25th, 2007 at 6:43 pm

[Quote](#)

先谢谢博主的回复，一直在找一种方便的方法将图片分离。

看来是最终替换法比较灵活，但我知道 `mod_rewrite` 是用来将用户提交的网址转换成服务器上的真实网址。

看了博主的回复好像它还有把网页执行的结果进行替换后再返回给浏览器的功能，是这样吗？

55

Michael says:

June 25th, 2007 at 11:00 pm

[Quote](#)

不是，只转换用户请求，对 `url` 进行 `rewrite`，进行重定向到新的 `url` 上，规则很灵活，建议仔细看看 `lighttpd` 或者 `apache` 的 `mod_rewrite` 文档，当然 `IIS` 也有类似的模块。

56

laoyao2k says:

June 25th, 2007 at 11:56 pm

[Quote](#)

我知道了，如果让客户浏览的网页代码里的图片地址是绝对地址，只能在开发时就写死了(对于静态页面)或用变量替换(对于动态页面更灵活些)，是这样吗？

我以为有更简单的方法呢，谢博主指点了。

57

马蜂不蛰 says:

July 24th, 2007 at 1:25 pm

[Quote](#)

请教楼主：

我正在搞一个医学教育视频资源在线预览的网站，只提供几分钟的视频预览，用 `swf` 格式，会员收看预览后线下可购买 DVD 光碟。

系统架构打算使用三台服务器：网页服务器、数据库服务器、视频服务器。

网页使用全部静态，数据库用 `SQL Server 2000`，CMS 是用 `ASP` 开发的。

会员数按十万级设计，不使用库表散列技术，请楼主给个建议，看看我的方案可行不？

58

Michael says:

July 24th, 2007 at 11:56 pm

[Quote](#)

这个数量级的应用好好配置优化一下服务器和代码，三台服务器完全没有问题，关键不是看整体会员数有多少，而是看同时在线的并发数有多少，并发不多就完全没有问题了，并发多的话，三台的这种架构还是有些问题的。

I mixi 技术架构

mixi.jp：使用开源软件搭建的可扩展 SNS 网站

总概关键点：

1, Mysql 切分, 采用 Innodb 运行

2, 动态 Cache 服务器 --

美国 Facebook.com, 中国 Yeejee.com, 日本 mixi.jp 均采用开源分布式缓存服务器 Memcache

3, 图片缓存和加

于敦德 2006-6-27

Mixi 目前是日本排名第三的网站, 全球排名 42, 主要提供 SNS 服务: 日记, 群组, 站内消息, 评论, 相册等等, 是日本最大的 SNS 网站。Mixi 从 2003 年 12 月份开始开发, 由现在它的 CTO - Batara Kesuma 一个人焊, 焊了四个月, 在 2004 年 2 月份开始上线运行。两个月后就注册了 1w 用户, 日访问量 60wPV。在随后的一年里, 用户增长到了 21w, 第二年, 增长到了 200w。到今年四月份已经增长到 370w 注册用户, 并且还在以每天 1.5w 人的注册量增长。这些用户中 70% 是活跃用户 (活跃用户: 三天内至少登录一次的用户), 平均每个用户每周在线时间为将近 3 个半小时。

下面我们来看它的技术架构。Mixi 采用开源软件作为架构的基础: Linux 2.6, Apache 2.0, MySQL, Perl 5.8, memcached, Squid 等等。到目前为止已经有 100 多台 MySQL 数据库服务器, 并且在以每月 10 多台的速度增长。Mixi 的数据库连接方式采用的是每次查询都进行连接, 而不是持久连接。数据库大多数是以 InnoDB 方式运行。Mixi 解决扩展问题主要依赖于对数据库的切分。

首先进行垂直切分, 按照表的内容将不同的表划分到不同的数据库中。然后是水平切分, 根据用户的 ID 将不同用户的内容再划分的不同的数据库中, 这是比较通常的做法, 也很管用。划分的关键还是在于应用中的实现, 需要将操作封装在数据层, 而尽量不影响业务层。当然完全不改变逻辑层也不可能, 这时候最能检验以前的设计是否到位, 如果以前设计的不错, 那创建连接的时候传个表名, 用户 ID 进去差不多就解决问题了, 而以前如果 sql 代码到处飞, 或者数据层封装的不太好的话那就累了。

这样做了以后并不能从根本上解决问题, 尤其是对于像 mixi 这种 SNS 网站, 页面上往往需要引用大量的用户信息, 好友信息, 图片, 文章信息, 跨表, 跨库操作相当多。这个时候就需要发挥 memcached 的作用了, 用大内存把这些不变的数据全都缓存起来, 而当修改时就通知 cache 过期, 这样应用层基本上就可以解决大部分问题了, 只会有很小一部分请求穿透应用层, 用到数据库。Mixi 的经验是平均每个页面的加载时间在 0.02 秒左右 (当然根据页面大小情况不尽相似), 可以说明这种做法是行之有效的。Mixi 一共有 32 台机器上有缓存服务器, 每个 Cache Server 2G 内存, 这些 Cache Server 与 App Server 装在一起。因为 Cache Server 对 CPU 消耗不大, 而有了 Cache Server 的支援, App Server 对内存要求也不是太高, 所以可以和平共处, 更有效的利用资源。

<http://dbplus.blog.51cto.com/194965/33632>

I memcached+squid+apache deflate 解决网站大访问量问题

不许联想的 RSS 之前停了两天, 据说是因为服务器负荷不了技术人员建议给关了, 不输出 RSS 能减轻多少负载呢? 所以月光博客不干了, 出来给支了几招, 但对于个人博客可能管用, 对于流量更大的专业网站显然需要进一步的优化。途牛最近的访问量增长得比较快, 所以很多页面 load 比较慢。之前我们就一直使用 memcached 进行了缓存以减轻数据库的压力, 近期又对 sql 查询进行了优化, 数据库的性能得到了明显的改善。途牛有很大一部分资源是图片, 针对这个我们使用 squid 进行了缓存, 这部分还包括 js、css 等一些静态文件。由于我们又有社区, 用户的反馈比较多, 所以页面并没有使用缓存, 而是使用 Apache 的 deflate

模块进行压缩。技术实现都比较简单但非常实用，通过这几步优化，途牛在响应速度上有了不小的提高。

I FeedBurner: 基于 MySQL 和 JAVA 的可扩展 Web 应用

于敦德 2006-6-27

FeedBurner（以下简称 FB，呵呵）我想应该是大家耳熟能详的一个名字，在国内我们有一个同样的服务商，叫做 **FeedSky**。在 2004 年 7 月份，FB 的流量是 300kbps，托管是 5600 个源，到 2005 年 4 月份，流量已经增长到 5Mbps，托管了 47700 个源；到 2005 年 9 月份流量增长到 20M，托管了 109200 个源，而到 2006 年 4 月份，流量已经到了 115Mbps，270000 个源，每天点击量一亿次。

FB 的服务使用 Java 实现，使用了 Mysql 数据库。我们下面来看一下 FB 在发展的过程中碰到的问题，以及解决的方案。

在 2004 年 8 月份，FB 的硬件设备包括 3 台 Web 服务器，3 台应用服务器和两台数据库服务器，使用 DNS 轮循分布服务负载，将前端请求分布到三台 Web 服务器上。说实话，如果不考虑稳定性，给 5600 个源提供服务应该用不了这么多服务器。现在的问题是即使用了这么多服务器他们还是无法避免单点问题，单点问题将至少影响到 1/3 的用户。FB 采用了监控的办法来解决，当监控到有问题出现时及时重启来避免更多用户受到影响。FB 采用了 Cacti(<http://www.cacti.net>)和 Nagios(<http://www.nagios.org>)来做监控。

FB 碰到的第二个问题是访问统计和管理。可以想象，每当我们在 RSS 阅读器里点击 FB 发布的内容，都需要做实时的统计，这个工作量是多么的巨大。大量写操作将导致系统的效率急剧下降，如果是 Myisam 表的话还会导致表的死锁。FB 一方面采用异步写入机制，通过创建执行池来缓冲写操作；只对本日的数据进行实时统计，而以前的数据以统计结果形式存储，进而避免每次查看访问统计时的重复计算。所以每一天第一次访问统计信息时速度可能会慢，这个时候应该是 FB 在分析整理前一天的数据，而接下来的访问由于只针对当日数据进行分析，数据量小很多，当然也会快很多。FB 的 Presentation 是这样写，但我发现好像我的 FB 里并没有今天实时的统计，也许是我观察的不够仔细-_-!

现在第三个问题出现了，由于大多数的操作都集中在主数据库上，数据库服务器的读写出现了冲突，前面提到过 **Myiasm** 类型的数据库在写入的时候会锁表，这样就导致了读写的冲突。在开始的时候由于读写操作比较少这个问题可能并不明显，但现在已经到了不能忽视的程度。解决方案是平衡读写的负载，以及扩展 **HibernateDaoSupport**，区分只读与读写操作，以实现针对读写操作的不同处理。

。解决方案是使用内存做缓存，而非数据库，他们同样使用了我们前面推荐的 **memcached**，同时他们还使用了 **Ehcache**。现在是第四个问题：数据库全面负载过高。由于使用数据库做为缓存，同时数据库被所有的应用服务器共享，速度越来越慢，而这时数据库大小也到了 **Myisam** 的上限-4GB，FB 的同学们自己都觉得自己有点懒 (<http://ehcache.sourceforge.net/>)，一款基于 **Java** 的分布式缓存工具。

第五个问题：流行 **rss** 源带来大量重复请求，导致系统待处理请求的堆积。同时我们注意到在 **RSS** 源小图标有时候会显示有多少用户订阅了这一 **RSS** 源，这同样需要服务器去处理，而目前所有的订阅数都在同一时间进行计算，导致对系统资源的大量占用。解决方案，把计算时间错开，同时在晚间处理堆积下来的请求，但这仍然不够。

问题六：状态统计写入数据库又一次出问题了。越来越多的辅助数据（包括广告统计，文章点击统计，订阅统计）需要写入数据库，导致太多的写操作。解决方案：每天晚上处理完堆积下来的请求后对子表进行截断操作：

– **FLUSH TABLES; TRUNCATE TABLE ad_stats0;**

这样的操作对 **Master** 数据库是成功的，但对 **Slave** 会失败，正确的截断子表方法是：

– **ALTER TABLE ad_stats TYPE=MERGE UNION=(ad_stats1,ad_stats2);**

– **TRUNCATE TABLE ad_stats0;**

– **ALTER TABLE ad_stats TYPE=MERGE UNION=(ad_stats0,ad_stats1,ad_stats2);**

解决方案的另外一部分就是我们最常用的水平分割数据库。把最常用的表分出去，单独做集群，例如广告啊，订阅计算啊，

第七个问题，问题还真多，主数据库服务器的单点问题。虽然采用了 Master-Slave 模式，但主数据库 Master 和 Slave 都只有一台，当 Master 出问题的时候需要太长的时间进行 Myisam 的修复，而 Slave 又无法很快的切换成为 Master。FB 试了好多办法，最终的解决方案好像也不是非常完美。从他们的实验过程来看，并没有试验 Master-Master 的结构，我想 Live Journal 的 Master-Master 方案对他们来说应该有用，当然要实现 Master-Master 需要改应用，还有有些麻烦的。

第八个问题，停电！芝加哥地区的供电状况看来不是很好，不过不管好不好，做好备份是最重要的，大家各显神通吧。

这个 Presentation 好像比较偏重数据库，当然了，谁让这是在 Mysql Con 上的发言，不过总给人一种不过瘾的感觉。另外一个感觉，FB 的 NO 们一直在救火，没有做系统的分析和设计。

最后 FB 的运维总监 Joe Kottke 给了四点建议：

- 1、 监控网站数据库负载。
- 2、“explain”所有的 SQL 语句。
- 3、 缓存所有能缓存的东西。
- 4、 归档好代码。

最后，FB 用到的软件都不是最新的，够用就好，包括：Tomcat5.0, Mysql 4.1, Hibernate 2.1, Spring, DBCP。

I YouTube 的架构扩展

flyincat 发布于：2007-07-25 09:23

作者：[Fengng](#) | [English Version](#) 【可以转载，转载时务必以超链接形式标明文章原始出处和作者信息及[版权声明](#)】

网址：http://www.dbanotes.net/opensource/youtube_web_arch.html

在[西雅图扩展性的技术研讨会](#)上，YouTube 的 Cuong Do 做了关于 [YouTube Scalability](#) 的报告。视频内容在 Google Video 上有[\(地址\)](#)，可惜国内用户看不到。

[Kyle Cordes](#) 对这个视频中的内容做了[介绍](#)。里面有不少技术性的内容。值得分享一下。(Kyle Cordes 的介绍是本文的主要来源)

简单的说 YouTube 的数据流量, "一天的 YouTube 流量相当于发送 750 亿封电子邮件.", 2006 年中就有消息说每日 PV 超过 1 亿, 现在? 更夸张了, "每天有 10 亿次下载以及 6,5000 次上传", 真假姑且不论, 的确是超乎寻常的海量. 国内的互联网应用, 但从数据量来看, 怕是只有 51.com 有这个规模. 但技术上和 YouTube 就没法子比了.

Web 服务器

YouTube 出于开发速度的考虑, 大部分代码都是 Python 开发的。Web 服务器有部分是 Apache, 用 FastCGI 模式。对于视频内容则用 [Lighttpd](#)。据我所知, MySpace 也有部分服务器用 Lighttpd, 但量不大。YouTube 是 Lighttpd 最成功的案例。(国内用 Lighttpd 站点不多, [豆瓣](#)用的比较舒服。by [Fennng](#))

视频

视频的缩略图(Thumbnails)给服务器带来了很大的挑战。每个视频平均有 4 个缩略图, 而每个 Web 页面上更是有多个, 每秒钟因为这个带来的磁盘 IO 请求太大。YouTube 技术人员启用了单独的服务器群组来承担这个压力, 并且针对 Cache 和 OS 做了部分优化。另一方面, 缩略图请求的压力导致 Lighttpd 性能下降。通过 Hack Lighttpd 增加更多的 worker 线程很大程度解决了问题。而最新的解决方案是起用了 Google 的 BigTable, 这下子从性能、容错、缓存上都有更好表现。看人家这收购的, 好钢用在了刀刃上。

出于冗余的考虑, 每个视频文件放在一组迷你 Cluster 上, 所谓 "迷你 Cluster" 就是一组具有相同内容的服务器。最火的视频放在 CDN 上, 这样自己的服务器只需要承担一些"漏网"的随即访问即可。YouTube 使用简单、廉价、通用的硬件, 这一点和 Google 风格倒是一致。至于维护手段, 也都是常见的工具, 如 rsync, SSH 等, 只不过人家更手熟罢了。

数据库

YouTube 用 MySQL 存储元数据--用户信息、视频信息什么的。数据库服务器曾经一度遇到 SWAP 颠簸的问题，解决办法是删掉了 SWAP 分区！管用。

最初的 DB 只有 10 块硬盘，RAID 10，后来追加了一组 RAID 1。够省的。这一波 Web 2.0 公司很少有用 Oracle 的(我知道的只有 [Bebo](#), 参见这里)。在扩展性方面，路线也是和其他站点类似，复制，分散 IO。最终的解决之道是"分区", 这个不是数据库层面的表分区，而是业务层面的分区(在用户名字或者 ID 上做文章, 应用程序控制查找机制)

YouTube 也用 [Memcached](#)。

很想了解一下国内 Web 2.0 网站的数据信息, 有谁可以提供一点？

--EOF--

[回复](#) | [引用](#) | [收藏](#) | [推荐给朋友](#) | [推荐到群组](#) | 22 次阅读 | 

标签： [系统架构](#)

引用地址： <http://www.mtime.com/blog/trackback/460579/>

I 了解一下 Technorati 的后台数据库架构

作者：Fenng | [English Version](#) 【可以转载，转载时务必以超链接形式标明文章原始出处和作者信息及版权声明】

网址：http://www.dbanotes.net/web/technorati_db_arch.html

Technorati (现在被阻尼了，可能你访问不了)的 Dorion Carroll 在 2006 MySQL 用户会议上介绍了一些关于 Technorati 后台数据库架构的情况。

基本情况

目前处理着大约 10Tb 核心数据，分布在大约 20 台机器上。通过复制，多增加了 100Tb 数据，分布在 200 台机器上。每天增长的数据 1TB。通过 SOA 的运用，物理与逻辑的访问相隔离，似乎消除了数据库的瓶颈。值得一提的是，该扩展过程始终是利用普通的硬件与开源软件来完成的。毕竟，Web 2.0 站点都不是烧钱的主。从数据量来看，这绝对是一个相对比较大的 Web 2.0 应用。

Tag 是 Technorati 最为重要的数据元素。爆炸性的 Tag 增长给 Technorati 带来了不小的挑战。

2005 年 1 月的时候，只有两台数据库服务器，一主一从。到了 06 年一月份，已经是一主一从，6 台 MyISAM 从数据库用来对付查询，3 台 MyISAM 用作异步计算。

一些核心的处理方法：

1) 根据实体(tags/posttags))进行分区

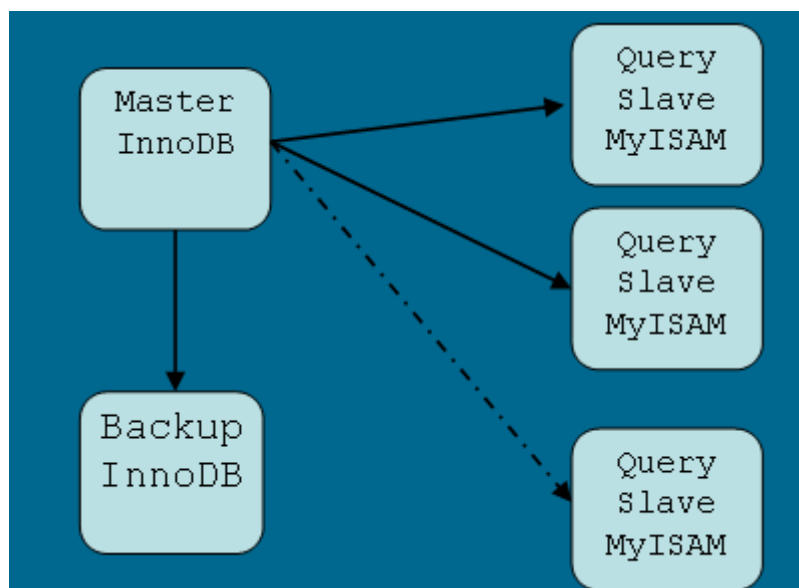
衡量数据访问方法，读和写的平衡。然后通过不同的维度进行分区。(Technorati 数据更新不会很多，否则会成为数据库灾难)

2) 合理利用 InnoDB 与 MyISAM

InnoDB 用于数据完整性/写性能要求比较高的应用。MyISAM 适合进行 OLAP 运算。物尽其用。

3) MySQL 复制

复制数据到从主数据库到辅数据库上,平衡分布查询与异步计算, 另外一个功能是提供冗余。 如图:



后记

拜读了一个藏袍的两篇大做([mixi.jp: 使用开源软件搭建的可扩展 SNS 网站](#) / [FeedBurner:基于 MySQL 和 JAVA 的可扩展 Web 应用](#)) 心痒难当, 顺藤摸瓜, 发现也有文档提及 Technorati, 赶紧照样学习一下. 几篇文档读罢, MySQL 的 可扩展性让我刮目相看.

或许,应该把注意力留一点给 MySQL 了 .

--End.

I Myspace 架构历程

亿万用户网站 MySpace 的成功秘密

© 文 / David F. Carr 译 / 罗小平

高速增长的访问量给社区网络的技术体系带来了巨大挑战。MySpace 的开发者优先多年来不断重构站点软件、数据库和存储系统, 以期与自身的成长同步——目前, 该站点月访问量已达 400 亿。绝大多数网站需要应对的流量都不及 MySpace 的一小部分, 但那些指望迈入庞大在线市场的人, 可以从 MySpace 的成长过程学到知识。

用户的烦恼

Drew, 是个来自达拉斯的 17 岁小伙子, 在他的 MySpace 个人资料页上, 可以看到他的袒胸照, 看样子是自己够着手拍的。他的好友栏全是漂亮姑娘和靓车的链接, 另外还说自己参加了学校田径队, 爱好吉他, 开一辆蓝色福特野马。

不过在用户反映问题的论坛里, 似乎他的火气很大。“赶紧弄好这该死的收件箱!”他大写了所有单词。使用 MySpace 的用户个人消息系统可以收发信息, 但当他要查看一条消息时, 页面却出现提示: “非常抱歉.....

消息错误”。

Drew 的抱怨说明 1.4 亿用户非常重视在线交流系统，这对 MySpace 来说是个好消息。但也恰是这点让 MySpace 成了全世界最繁忙的站点之一。

11 月，MySpace 的美国国内互联网用户访问流量首次超过 Yahoo。comScore Media Metrix 公司提供的资料显示，MySpace 当月访问量为 387 亿，而 Yahoo 是 380.5 亿。

显然，MySpace 的成长太快了——从 2003 年 11 月正式上线到现在不过三年。这使它很早就要面对只有极少数公司才会遇到的高可扩展性问题的严峻挑战。

事实上，MySpace 的 Web 服务器和数据库经常性超负荷，其用户频繁遭遇“意外错误”和“站点离线维护”等告示。包括 Drew 在内的 MySpace 用户经常无法收发消息、更新个人资料或处理其他日常事务，他们不得不在论坛抱怨不停。

尤其是最近，MySpace 可能经常性超负荷。因为 Keynote Systems 公司性能监测服务机构负责人 Shawn White 说，“难以想象，在有些时候，我们发现 20% 的错误日志都来自 MySpace，有时候甚至达到 30% 以至 40%……而 Yahoo、Salesforce.com 和其他提供商用服务的站点，从来不会出现这样的数字。”他告诉我们，其他大型站点的日错误率一般就 1% 多点。

顺便提及，MySpace 在 2006 年 7 月 24 号晚上开始了长达 12 小时的瘫痪，期间只有一个可访问页面——该页面解释说位于洛杉矶的主数据中心发生故障。为了让大家耐心等待服务恢复，该页面提供了用 Flash 开发的派克人 (Pac-Man) 游戏。Web 站点跟踪服务研究公司总经理 Bill Tancer 说，尤其有趣的是，MySpace 瘫痪期间，访问量不降反升，“这说明了人们对 MySpace 的痴迷——所有人都拥在它的门口等着放行”。

现 Nielsen Norman Group 咨询公司负责人、原 Sun Microsystems 公司工程师，因在 Web 站点方面的评论而闻名的 Jakob Nielsen 说，MySpace 的系统构建方法显然与 Yahoo、eBay 以及 Google 都不相同。和很多观察家一样，他相信 MySpace 对其成长速度始料未及。“虽然我不认为他们必须在计算机科学领域全面创新，但他们面对的的确是一个巨大的科学难题。”他说。

MySpace 开发人员已经多次重构站点软件、数据库和存储系统，以满足爆炸性的成长需要，但此工作永不会停息。“就像粉刷金门大桥，工作完成之时，就是重新来过之日。”（译者注：意指工人从桥头开始粉刷，当到达桥尾时，桥头涂料已经剥落，必须重新开始）MySpace 技术副总裁 Jim Benedetto 说。

既然如此，MySpace 的技术还有何可学之处？因为 MySpace 事实上已经解决了很多系统扩展性问题，才能走到今天。

Benedetto 说他的项目组有很多教训必须总结，他们仍在学习，路漫漫而修远。他们当前需要改进的工作包括实现更灵活的数据缓存系统，以及为避免再次出现类似 7 月瘫痪事件的地理上分布式架构。

背景知识

MySpace 目前的努力方向是解决扩展性问题，但其领导人最初关注的是系统性能。

3 年多前，一家叫做 Intermix Media（早先叫 eUniverse。这家公司从事各类电子邮件营销和网上商务）的公司推出了 MySpace。而其创建人是 Chris DeWolfe 和 Tom Anderson，他们原来也有一家叫做 ResponseBase 的电子邮件营销公司，后于 2002 年出售给 Intermix。据 Brad Greenspan（Intermix 前 CEO）运作的一个网站披露，ResponseBase 团队为此获得 2 百万美金外加分红。Intermix 是一家颇具侵略性的互联网商务公司——部分做法可能有点过头。2005 年，纽约总检察长 Eliot Spitzer——现在是纽约州长——起诉 Intermix 使用恶意广告软件推广业务，Intermix 最后以 790 万美元的代价达成和解。

2003 年，美国国会通过《反垃圾邮件法》（CAN-SPAM Act），意在控制滥发邮件的营销行为。Intermix 领导人 DeWolfe 和 Anderson 意识到新法案将严重打击公司的电子邮件营销业务，“因此必须寻找新的方向。”受聘于 Intermix 负责重写公司邮件营销软件的 Duc Chau 说。

当时有个叫 Friendster 的交友网站，Anderson 和 DeWolfe 很早就是它的会员。于是他们决定创建自己的网上社区。他们去除了 Friendster 在用户自我表述方面的诸多限制，并重点突出音乐（尤其是重金属乐），希望以此吸引用户。Chau 使用 Perl 开发了最初的 MySpace 版本，运行于 Apache Web 服务器，后台使用 MySQL 数据库。但它没有通过终审，因为 Intermix 的多数开发人员对 ColdFusion（一个 Web 应用程序环境，最初由 Allaire 开发，现为 Adobe 所有）更为熟悉。因此，最后发布的产品采用 ColdFusion 开发，运行在 Windows 上，并使用 MS SQL Server 作为数据库服务器。

Chau 就在那时离开了公司，将开发工作交给其他人，包括 Aber Whitcomb（Intermix 的技术专家，现在是 MySpace 技术总监）和 Benedetto（MySpace 现技术副总裁，大概于 MySpace 上线一个月后加入）。

MySpace 上线的 2003 年，恰恰是 Friendster 在满足日益增长的用户需求问题上遭遇麻烦的时期。在财富杂志最近的一次采访中，Friendster 总裁 Kent Lindstrom 承认他们的服务出现问题选错了时候。那时，Friendster 传输一个页面需要 20 到 30 秒，而 MySpace 只需 2 到 3 秒。

结果，Friendster 用户开始转投 MySpace，他们认为后者更为可靠。

今天，MySpace 无疑已是社区网站之王。社区网站是指那些帮助用户彼此保持联系、通过介绍或搜索、基于共同爱好或教育经历交友的 Web 站点。在这个领域比较有名的还有最初面向大学生的 Facebook、侧重职业交流的 LinkedIn，当然还少不了 Friendster。MySpace 宣称自己是“下一代门户”，强调内容的丰富多彩（如音乐、趣事和视频等）。其运作方式颇似一个虚拟的夜总会——为未成年人在边上安排一个果汁吧，而显著位置则是以性为目的的约会，和寻找刺激派对气氛的年轻人的搜索服务。

用户注册时，需要提供个人基本信息，主要包括籍贯、性取向和婚姻状况。虽然 MySpace 屡遭批评，指其为网上性犯罪提供了温床，但对于未成年人，有些功能还是不予提供的。

MySpace 的个人资料页上表述自己的方式很多，如文本式“关于本人”栏、选择加载入 MySpace 音乐播放器的歌曲，以及视频、交友要求等。它还允许用户使用 CSS（一种 Web 标准格式语言，用户以此可设置页面元素的字体、颜色和页面背景图像）自由设计个人页面，这也提升了人气。不过结果是五花八门——很多用户的页面布局粗野、颜色迷乱，进去后找不到东南西北，不忍卒读；而有些人则使用了专业设计的模版（可阅读《Too Much of a Good Thing?》第 49 页），页面效果很好。

在网站上线 8 个月后，开始有大量用户邀请朋友注册 MySpace，因此用户量大增。“这就是网络的力量，这种趋势一直没有停止。”Chau 说。

拥有 Fox 电视网络和 20th Century Fox 影业公司的媒体帝国——新闻集团，看到了他们在互联网用户中的机会，于是在 2005 年斥资 5.8 亿美元收购了 MySpace。新闻集团董事局主席 Rupert Murdoch 最近向一个投资团透露，他认为 MySpace 目前是世界主要 Web 门户之一，如果现在出售 MySpace，那么可获 60 亿美元——这比 2005 年收购价格的 10 倍还多！新闻集团还惊人地宣称，MySpace 在 2006 年 7 月结束的财政年度里总收入约 2 亿美元，而且预期在 2007 年度，Fox Interactive 公司总收入将达到 5 亿美元，其中 4 亿来自 MySpace。

然而 MySpace 还在继续成长。12 月份，它的注册账户达到 1.4 亿，而 2005 年 11 月时不过 4 千万。当然，这个数字并不等于真实的用户个体数，因为有些人可能有多个帐号，而且个人资料也表明有些是乐队，或者是虚构的名字，如波拉特（译者注：喜剧电影《Borat》主角），还有像 Burger King（译者注：美国最大的汉堡连锁集团）这样的品牌名。

当然，这么多的用户不停发布消息、撰写评论或者更新个人资料，甚至一些人整天都泡在 Space 上，必然给 MySpace 的技术工作带来前所未有的挑战。而传统新闻站点的绝大多数内容都是由编辑团队整理后主动提供给用户消费，它们的内容数据库通常可以优化为只读模式，因为用户评论等引起的增加和更新操作很少。而 MySpace 是由用户提供内容，数据库很大比例的操作都是插入和更新，而非读取。

浏览 MySpace 上的任何个人资料时，系统都必须先查询数据库，然后动态创建页面。当然，通过数据缓存，可以减轻数据库的压力，但这种方案必须解决原始数据被用户频繁更新带来的同步问题。

MySpace 的站点架构已经历了 5 个版本——每次都是用户数达到一个里程碑后，必须做大量的调整和优化。Benedetto 说，“但我们始终跟不上形势的发展速度。我们重构重构再重构，一步步挪到今天”。

尽管 MySpace 拒绝了正式采访，但 Benedetto 在参加 11 月于拉斯维加斯召开的 SQL Server Connections 会议时还是回答了 Baseline 的问题。本文的不少技术信息还来源于另一次重要会议——Benedetto 和他的老板——技术总监 Whitcomb 今年 3 月出席的 Microsoft MIX Web 开发者大会。

据他们讲，MySpace 很多大的架构变动都发生在 2004 和 2005 年早期——用户数在当时从几十万迅速攀升到了几百万。

在每个里程碑，站点负担都会超过底层系统部分组件的最大载荷，特别是数据库和存储系统。接着，功能出现问题，用户失声尖叫。最后，技术团队必须为此修订系统策略。

虽然自 2005 年早期，站点账户数超过 7 百万后，系统架构到目前为止保持了相对稳定，但 MySpace 仍然在为 SQL Server 支持的同时连接数等方面继续攻坚，Benedetto 说，“我们已经尽可能把事情做到最好”。

里程碑一：50 万账户

按 Benedetto 的说法，MySpace 最初的系统很小，只有两台 Web 服务器和一个数据库服务器。那时使用的是 Dell 双 CPU、4G 内存的系统。

单个数据库就意味着所有数据都存储在一个地方，再由两台 Web 服务器分担处理用户请求的工作量。但就像 MySpace 后来的几次底层系统修订时的情况一样，三服务器架构很快不堪重负。此后一个时期内，MySpace 基本是通过添置更多 Web 服务器来对付用户暴增问题的。

但到在 2004 年早期，MySpace 用户数增长到 50 万后，数据库服务器也已开始汗流浹背。

但和 Web 服务器不同，增加数据库可没那么简单。如果一个站点由多个数据库支持，设计者必须考虑的是，如何在保证数据一致性的前提下，让多个数据库分担压力。

在第二代架构中，MySpace 运行在 3 个 SQL Server 数据库服务器上——一个为主，所有的新数据都向它提交，然后由它复制到其他两个；另两个全力向用户供给数据，用以在博客和个人资料栏显示。这种方式在一段时间内效果很好——只要增加数据库服务器，加大硬盘，就可以应对用户数和访问量的增加。

里程碑二：1-2 百万账户

MySpace 注册数到达 1 百万至 2 百万区间后，数据库服务器开始受制于 I/O 容量——即它们存取数据的速度。而当时才是 2004 年中，距离上次数据库系统调整不过数月。用户的提交请求被阻塞，就像千人乐迷要挤进只能容纳几百人的夜总会，站点开始遭遇“主要矛盾”，Benedetto 说，这意味着 MySpace 永远都会轻度落后于用户需求。

“有人花 5 分钟都无法完成留言，因此用户总是抱怨说网站已经完蛋了。”他补充道。

这一次的数据库架构按照垂直分割模式设计，不同的数据库服务于站点的不同功能，如登录、用户资料和博客。于是，站点的扩展性问题看似又可以告一段落了，可以歇一阵子。

垂直分割策略利于多个数据库分担访问压力，当用户要求增加新功能时，MySpace 将投入新的数据库予以支持它。账户到达 2 百万后，MySpace 还从存储设备与数据库服务器直接交互的方式切换到 SAN（Storage Area Network，存储区域网络）——用高带宽、专门设计的网络将大量磁盘存储设备连接在一起，而数据

库连接到 SAN。这项措施极大提升了系统性能、正常运行时间和可靠性，Benedetto 说。

里程碑三：3 百万账户

当用户继续增加到 3 百万后，垂直分割策略也开始难以为继。尽管站点的各个应用被设计得高度独立，但有些信息必须共享。在这个架构里，每个数据库必须有各自的用户表副本——MySpace 授权用户的电子花名册。这就意味着一个用户注册时，该条账户记录必须在 9 个不同数据库上分别创建。但在个别情况下，如果其中某台数据库服务器临时不可到达，对应事务就会失败，从而造成账户非完全创建，最终导致此用户的该项服务无效。

另外一个问题是，个别应用如博客增长太快，那么专门为它服务的数据库就有巨大压力。

2004 年中，MySpace 面临 Web 开发者称之为“向上扩展”对“向外扩展”（译者注：Scale Up 和 Scale Out，也称硬件扩展和软件扩展）的抉择——要么扩展到更大更强、也更昂贵的服务器上，要么部署大量相对便宜的服务器来分担数据库压力。一般来说，大型站点倾向于向外扩展，因为这将让它们得以保留通过增加服务器以提升系统能力的后路。

但成功地向外扩展架构必须解决复杂的分布式计算问题，大型站点如 Google、Yahoo 和 Amazon.com，都必须自行研发大量相关技术。以 Google 为例，它构建了自己的分布式文件系统。

另外，向外扩展策略还需要大量重写原来软件，以保证系统能在分布式服务器上运行。“搞不好，开发人员的所有工作都将白费”，Benedetto 说。

因此，MySpace 首先将重点放在了向上扩展上，花费了大约 1 个半月时间研究升级到 32CPU 服务器以管理更大数据库的问题。Benedetto 说，“那时候，这个方案看似可能解决一切问题。”如稳定性，更棒的是对现有软件几乎没有改动要求。

糟糕的是，高端服务器极其昂贵，是购置同样处理能力和内存速度的多台服务器总和的很多倍。而且，站点架构师预测，从长期来看，即便是巨型数据库，最后也会不堪重负，Benedetto 说，“换句话讲，只要增长趋势存在，我们最后无论如何都要走上向外扩展的道路。”

因此，MySpace 最终将目光移到分布式计算架构——它在物理上分布的众多服务器，整体必须逻辑上等同于单台机器。拿数据库来说，就不能再像过去那样将应用拆分，再以不同数据库分别支持，而必须将整个站点看作一个应用。现在，数据库模型里只有一个用户表，支持博客、个人资料和其他核心功能的数据都存储在相同数据库。

既然所有的核心数据逻辑上都组织到一个数据库，那么 MySpace 必须找到新的办法以分担负荷——显然，运行在普通硬件上的单个数据库服务器是无能为力的。这次，不再按站点功能和应用分割数据库，MySpace 开始将它的用户按每百万一组分割，然后将各组的全部数据分别存入独立的 SQL Server 实例。目前，MySpace 的每台数据库服务器实际运行两个 SQL Server 实例，也就是说每台服务器服务大约 2 百万用户。Benedetto 指出，以后还可以按照这种模式以更小粒度划分架构，从而优化负荷分担。

当然，还是有一个特殊数据库保存了所有账户的名称和密码。用户登录后，保存了他们其他数据的数据库再接管服务。特殊数据库的用户表虽然庞大，但它只负责用户登录，功能单一，所以负荷还是比较容易控制的。

里程碑四：9 百万到 1 千 7 百万账户

2005 年早期，账户达到 9 百万后，MySpace 开始用 Microsoft 的 C# 编写 ASP.NET 程序。C# 是 C 语言的最新派生语言，吸收了 C++ 和 Java 的优点，依托于 Microsoft .NET 框架（Microsoft 为软件组件化和分布式计算而设计的模型架构）。ASP.NET 则由编写 Web 站点脚本的 ASP 技术演化而来，是 Microsoft 目前主推的 Web 站点编程环境。

可以说是立竿见影，MySpace 马上就发现 ASP.NET 程序运行更有效率，与 ColdFusion 相比，完成同样任务需消耗的处理能力更小。据技术总监 Whitcomb 说，新代码需要 150 台服务器完成的工作，如果用 ColdFusion 则需要 246 台。Benedetto 还指出，性能上升的另一个原因可能是在变换软件平台，并用新语

言重写代码的过程中，程序员复审并优化了一些功能流程。

最终，MySpace 开始大规模迁移到 ASP.NET。即便剩余的少部分 ColdFusion 代码，也从 Cold-Fusion 服务器搬到了 ASP.NET，因为他们得到了 BlueDragon.NET（乔治亚州阿尔法利塔 New Atlanta Communications 公司的产品，它能将 ColdFusion 代码自动重新编译到 Microsoft 平台）的帮助。

账户达到 1 千万时，MySpace 再次遭遇存储瓶颈问题。SAN 的引入解决了早期一些性能问题，但站点目前的要求已经开始周期性超越 SAN 的 I/O 容量——即它从磁盘存储系统读写数据的极限速度。

原因之一是每数据库 1 百万账户的分割策略，通常情况下的确可以将压力均分到各台服务器，但现实并非一成不变。比如第七台账户数据库上线后，仅仅 7 天就被塞满了，主要原因是佛罗里达一个乐队的歌迷疯狂注册。

某个数据库可能因为任何原因，在任何时候遭遇主要负荷，这时，SAN 中绑定到该数据库的磁盘存储设备簇就可能过载。“SAN 让磁盘 I/O 能力大幅提升了，但将它们绑定到特定数据库的做法是错误的。”Benedetto 说。

最初，MySpace 通过定期重新分配 SAN 中数据，以让其更为均衡的方法基本解决了这个问题，但这是一个人工过程，“大概需要两个人全职工作。”Benedetto 说。

长期解决方案是迁移到虚拟存储体系上，这样，整个 SAN 被当作一个巨型存储池，不再要求每个磁盘为特定应用服务。MySpace 目前采用了一种新型 SAN 设备——来自加利福尼亚州弗里蒙特的 3PARdata。

在 3PAR 的系统里，仍能在逻辑上按容量划分数据存储，但它不再被绑定到特定磁盘或磁盘簇，而是散布于大量磁盘。这就使均分数据访问负荷成为可能。当数据库需要写入一组数据时，任何空闲磁盘都可以马上完成这项工作，而不再像以前那样阻塞在可能已经过载的磁盘阵列处。而且，因为多个磁盘都有数据副本，读取数据时，也不会使 SAN 的任何组件过载。

当 2005 年春天账户数达到 1 千 7 百万时，MySpace 又启用了新的策略以减轻存储系统压力，即增加数据缓存层——位于 Web 服务器和数据库服务器之间，其唯一职能是在内存中建立被频繁请求数据对象的副本，如此一来，不访问数据库也可以向 Web 应用供给数据。换句话说，100 个用户请求同一份资料，以前需要查询数据库 100 次，而现在只需 1 次，其余都可从缓存数据中获得。当然如果页面变化，缓存的数据必须从内存擦除，然后重新从数据库获取——但在此之前，数据库的压力已经大大减轻，整个站点的性能得到提升。

缓存区还为那些不需要记入数据库的数据提供了驿站，比如为跟踪用户会话而创建的临时文件——Benedetto 坦言他需要在这方面补课，“我是数据库存储狂热分子，因此我总是想着将万事万物都存到数据库。”但将像会话跟踪这类的数据也存到数据库，站点将陷入泥沼。

增加缓存服务器是“一开始就应该做的事情，但我们成长太快，以致于没有时间坐下来好好研究这件事情。”Benedetto 补充道。

里程碑五：2 千 6 百万账户

2005 年中期，服务账户数达到 2 千 6 百万时，MySpace 切换到了还处于 beta 测试的 SQL Server 2005。转换何太急？主流看法是 2005 版支持 64 位处理器。但 Benedetto 说，“这不是主要原因，尽管这也很重要；主要还是因为我们对内存的渴求。”支持 64 位的数据库可以管理更多内存。

更多内存就意味着更高的性能和更大的容量。原来运行 32 位版本的 SQL Server 服务器，能同时使用的内存最多只有 4G。切换到 64 位，就好像加粗了水管的直径。升级到 SQL Server 2005 和 64 位 Windows Server 2003 后，MySpace 每台服务器配备了 32G 内存，后于 2006 年再次将配置标准提升到 64G。

意外错误

如果没有对系统架构的历次修改与升级，MySpace 根本不可能走到今天。但是，为什么系统还经常吃撑着了？很多用户抱怨的“意外错误”是怎么引起的呢？

原因之一是 MySpace 对 Microsoft 的 Web 技术的应用已经进入连 Microsoft 自己也才刚刚开始探索的领域。比如 11 月，超出 SQL Server 最大同时连接数，MySpace 系统崩溃。Benedetto 说，这类可能引发系统崩溃的情况大概三天才会出现一次，但仍然过于频繁了，以致惹人恼怒。一旦数据库罢工，“无论这种情况什么时候发生，未缓存的数据都不能从 SQL Server 获得，那么你就必然看到一个‘意外错误’提示。”他解释说。去年夏天，MySpace 的 Windows 2003 多次自动停止服务。后来发现是操作系统一个内置功能惹的祸——预防分布式拒绝服务攻击（黑客使用很多客户机向服务器发起大量连接请求，以致服务器瘫痪）。MySpace 和其他很多顶级大站点一样，肯定会经常遭受攻击，但它应该从网络级而不是依靠 Windows 本身的功能来解决问题——否则，大量 MySpace 合法用户连接时也会引起服务器反击。

“我们花了大约一个月时间寻找 Windows 2003 服务器自动停止的原因。”Benedetto 说。最后，通过 Microsoft 的帮助，他们才知道该怎么通知服务器：“别开枪，是友军。”

紧接着是在去年 7 月某个周日晚上，MySpace 总部所在地洛杉矶停电，造成整个系统停运 12 小时。大型 Web 站点通常要在地理上分布配置多个数据中心以预防单点故障。本来，MySpace 还有其他两个数据中心以应对突发事件，但 Web 服务器都依赖于部署在洛杉矶的 SAN。没有洛杉矶的 SAN，Web 服务器除了恳求你耐心等待，不能提供任何服务。

Benedetto 说，主数据中心的可靠性通过下列措施保证：可接入两张不同电网，另有后备电源和一台储备有 30 天燃料的发电机。但在这次事故中，不仅两张电网失效，而且在切换到备份电源的过程中，操作员烧掉了主动力线路。

2007 年中，MySpace 在另两个后备站点上也建设了 SAN。这对分担负荷大有帮助——正常情况下，每个 SAN 都能负担三分之一的数据访问量。而在紧急情况下，任何一个站点都可以独立支撑整个服务，Benedetto 说。

MySpace 仍然在为提高稳定性奋斗，虽然很多用户表示了足够信任且能原谅偶现的错误页面。

“作为开发人员，我憎恶 Bug，它太气人了。”Dan Tanner 这个 31 岁的德克萨斯软件工程师说，他通过 MySpace 重新联系到了高中和大学同学。“不过，MySpace 对我们的用处很大，因此我们可以原谅偶发的故障和错误。”Tanner 说，如果站点某天出现故障甚至崩溃，恢复以后他还是会继续使用。

这就是为什么 Drew 在论坛里咆哮时，大部分用户都告诉他应该保持平静，如果等几分钟，问题就会解决的原因。Drew 无法平静，他写道，“我已经两次给 MySpace 发邮件，而它说一小时前还是正常的，现在出了点问题……完全是一堆废话。”另一个用户回复说，“毕竟它是免费的。”Benedetto 坦承 100% 的可靠性不是他的目标。“它不是银行，而是一个免费的服务。”他说。

换句话说，MySpace 的偶发故障可能造成某人最后更新的个人资料丢失，但并不意味着网站弄丢了用户的钱财。“关键是要认识到，与保证站点性能相比，丢失少许数据的故障是可接受的。”Benedetto 说。所以，MySpace 甘冒丢失 2 分钟到 2 小时内任意点数据的危险，在 SQL Server 配置里延长了“checkpoint”操作——它将待更新数据永久记录到磁盘——的间隔时间，因为这样做可以加快数据库的运行。

Benedetto 说，同样，开发人员还经常在几个小时内就完成构思、编码、测试和发布全过程。这有引入 Bug 的风险，但这样做可以更快实现新功能。而且，因为进行大规模真实测试不具可行性，他们的测试通常是在仅以部分活跃用户为对象，且用户对软件新功能和改进不知就里的情况下进行的。因为事实上不可能做真实的加载测试，他们做的测试通常都是针对站点。

“我们犯过大量错误，”Benedetto 说，“但到头来，我认为我们做对的还是比做错的多。”

Trackback: <http://tb.blog.csdn.net/TrackBack.aspx?PostId=1536222>

I eBay 的数据量

作者: Fenng | [English Version](#) 【可以转载, 转载时务必以超链接形式标明文章原始出处和作者信息及版权声明】

网址: http://www.dbanotes.net/database/ebay_storage.html

作为电子商务领头羊的 eBay 公司, 数据量究竟有多大? 很多朋友可能都会对这个很感兴趣。在这一篇 [Web 2.0: How High-Volume eBay Manages Its Storage](#)(从+1 GB/1 min 得到的线索) 报道中, eBay 的存储主管 Paul Strong 对数据量做了一些介绍, 管中窥豹, 这些数据也给我们一个参考。

站点处理能力

- 平均每天的 PV 超过 10 亿 ;
- 每秒钟交易大约 1700 美元的商品 ;
- 每分钟卖出一辆车 A ;
- 每秒钟卖出一件汽车饰品或者配件 ;
- 每两分钟卖出一件钻石首饰 ;
- 6 亿商品, 2 亿多注册用户; 超过 130 万人把在 eBay 上做生意看作是生活的一部分。

在这样高的压力下, 可靠性达到了 99.94%, 也就是说每年 5 个小时多一点的服务不可用。从业界消息来看, 核心业务的可用性要比这个高。

数据存储工程组控制着 eBay 的 2PB (1Petabyte=1000Terabytes) 可用空间。这是一个什么概念, 对比一下 Google 的存储就知道了。每周就要分配 10T 数据出去, 稍微算一下, 一分钟大约使用 1G 的数据空间。

计算能力

eBay 使用一套传统的网格计算系统。该系统的一些特征数据:

- 170 台 Win2000/Win2003 服务器;
- 170 台 Linux (RHES3) 服务器;
- 三个 Solaris 服务器: 为 QA 构建与部署 eBay.com; 编译优化 Java / C++ 以及其他 Web 元素 ;
- Build 整个站点的时间: 过去是 10 个小时, 现在是 30 分钟;
- 在过去的 2 年半, 有 200 万次 Build, 很可怕的数字。

存储硬件

每个供货商都必须通过严格的测试才有被选中的可能, 这些厂家或产品如下:

- 交换机: Brocade
- 网管软件: IBM Tivoli

- NAS: Netapp (占总数据量的 5%, 2P*0.05, 大约 100 T)
- 阵列存储: HDS (95%, 这一份投资可不小, HDS 不便宜, EMC 在 eBay 是出局者) 负载均衡与 Failover: Resonate ;

搜索功能: Thunderstone indexing system ;

数据库软件: Oracle 。大多数 DB 都有 4 份拷贝。数据库使用的服务器 Sun E10000。另外据我所知, eBay 购买了 Quest SharePlex 全球 Licence 用于数据复制。

应用服务器

应用服务器有哪些特点呢?

- ~~使用单一的两层架构(这一点有点疑问, 看来是自己写的应用服务器)~~
- ~~330 万行的 C++ ISAPI DLL (二进制文件有 150M)~~
- 数百名工程师进行开发
- ~~每个类的方法已经接近编译器的限制~~

非常有意思, 根据 eWeek 的该篇文档, 昨天还有上面这段划掉的内容, 今天上去发现已经修改了:

架构

- 高分布式
- 拍卖站点是基于 Java 的, 搜索的架构是用 C++ 写的
- 数百名工程师进行开发, 所有的工作都在同样的代码环境下进行

可能是被采访者看到 eWeek 这篇报道, 联系了采访者进行了更正。我还有点奇怪原来"两层"架构的说法。

其他信息

- 集中化存储应用程序日志;
- 全局计费: 实时的与第三方应用集成(就是 eBay 自己的 PayPal 吧?)
- 业务事件流: 使用统一的高效可靠消息队列. 并且使用 Cookie-cutter 模式用于优化用户体验(这似乎是大型电子商务站点普遍使用的用于提高用户体验的手法)。

后记

零散作了一点流水帐。作为一个 DBA, 或许有一天也有机会面对这样的数据量。到那一天, 再回头看这一篇电子垃圾。

更新：更详细信息请参考：[Web 2.0: How High-Volume eBay Manages Its Storage](#)。可能处于 Cache 的问题，好几个人看到的原文内容有差异

--EOF--

I eBay 的应用服务器规模

作者：Fenng | [English Version](#) 【可以转载，转载时务必以超链接形式标明文章原始出处和作者信息及版权声明】

网址：http://www.dbanotes.net/web/ebay_application_server.html

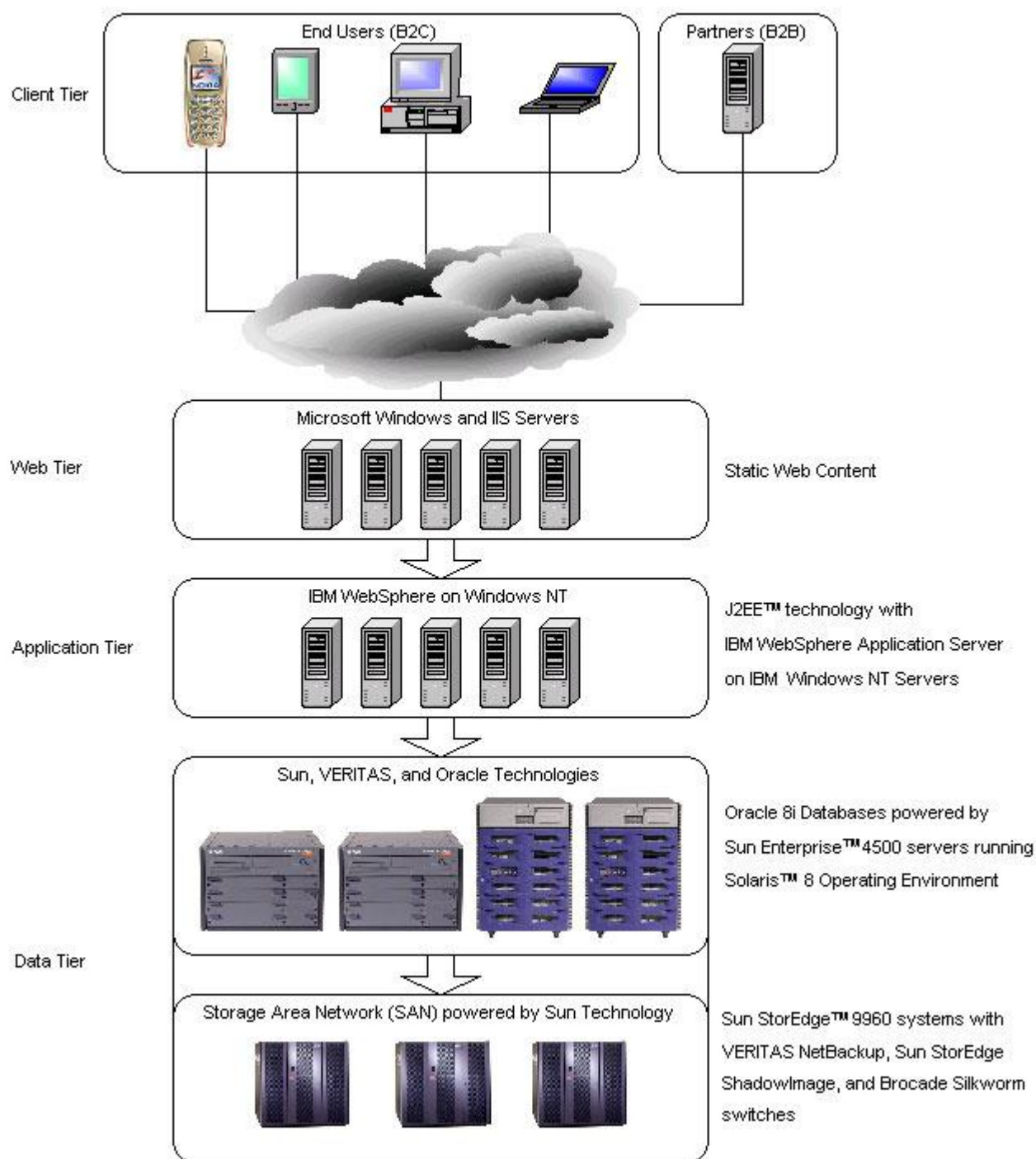
前面我在《eBay 的数据量》中介绍了一些道听途说来的关于互联网巨头 eBay 服务器架构的信息，不过还缺了一点关键数据。

在 Oracle 站点上的一篇题为 [The eBay Global Platform and Oracle 10g JDBC](#) 的白皮书，有能看到一些数据。

在 2004 年的时候，eBay 的应用服务器采用了 IBM WebSphere，部署在 WinNT 上，硬件是 Intel 双 CPU 奔腾服务器。**服务器数量是 2400 台**。在《eBay 的数据量》中我们知道，eBay 的是集中式处理 Log 的，每天会有 2T 的 Log 数据产生，现在只会更多。这些应用服务器分成不同的组，通过一个统一的 DAL(database access layer) 逻辑层访问 135 个数据库节点。

这篇白皮书已经发布了两年，相信在这两年的时间里，服务器规模又会扩大了许多。

eBay 的 SOA 架构 V3 示意图如下：



这个图来自[这里](#)

以前我写的《[这些大网站都用什么操作系统与 Web 服务器 ?](#)》，还有网友质疑 eBay 的服务器不是 WinNT，现在倒是间接证明了 Web 服务器的确是 Windows 。

I eBay 的数据库分布扩展架构

作者: Fenng | [English Version](#) 【可以转载，转载时务必以超链接形式标明文章原始出处和作者信息及版

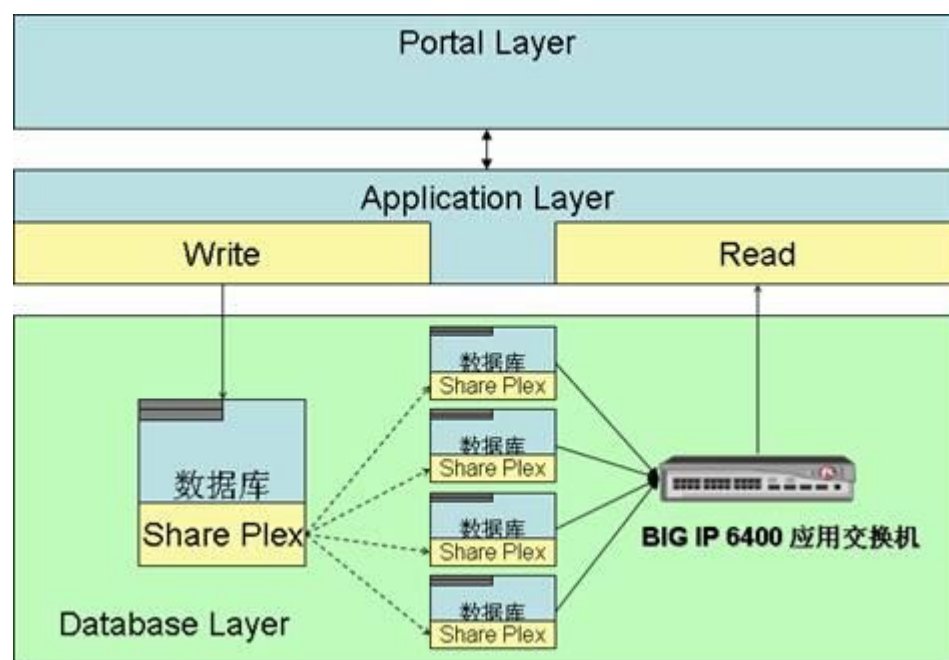
权声明】

网址: http://www.dbanotes.net/database/ebay_database_scale_out.html

在过去的 Blog 中, 我(插一嘴: 这里的"我" 如果替换成 "Fenng" 似乎有些自恋, 也不是我喜欢的行文语气, 可发现转贴不留名的行为太多了, 他大爷的)曾经介绍过 《eBay 的应用服务器规模》, 也介绍过 《eBay 的数据量》, 在这篇文章中提到过 "eBay 购买了 Quest Share Plex 全球 Licence 用于数据复制", 这个地方其实没有说开来。

对于 eBay 这样超大规模的站点来说, 瓶颈往往最容易在数据库服务器上产生, 必定有一部分数据(比如交易记录这样不容易水平分割的数据)容易带来大量的读操作, 而不管用什么存储, 能承担的 IO 能力是有限的。所以, 如果有效的分散 IO 的承载能力就是一个很有意义的事情。

经过互联网考古学不断挖掘, 路路续续又现了一些蛛丝马迹能够多少说明一些问题。客观事实加上主观想象, 简单的描述一下。见下图:



通过 Quest 公司的 Share Plex 近乎实时的复制数据到其他数据库节点, F5 通过特定的模块检查数据库状态, 并进行负载均衡, IO 成功的做到了分布, 读写分离, 而且极大的提高了可用性。F5 真是一家很有创新性的公司, 虽然从这个案例来说, 技术并无高深之处, 但方法巧妙, 整个方案浑然一体。

F5 公司专门为 Oracle 9i 数据库开发了专用的健康检查模块, 通过调用 F5 专有的扩展应用校验(EAV)进程, F5 能够随时得到 Oracle 9i 数据库的应用层服务能力而不是其他的负载均衡设备所采用的 ICMP/TCP 层进行健康检查。

这个图来自一篇《F5 助力 eBay 数据库服务器负载均衡》的软文, 真是一篇很好的软文, 国外恐怕不会出现这样"含金量"极高的东西。

当然, 这个技术架构可不算便宜。Quest 的 Share Plex License 很贵, 而且, 对于每个结点来说, 都需要数据库 License 与硬件费用。但优点也很多: 节省了维护成本; 数据库层面的访问也能做到 SOA; 高可用性。

国内的一些厂商比较喜欢给客户推荐存储级别的解决方案。通过存储底层复制来解决数据分布以及灾备问题。这个思路似乎太传统了, 对于互联网企业来说多少有点过时。

I 从 LiveJournal 后台发展看大规模网站性能优化方法

于敦德 2006-3-16

一、LiveJournal 发展历程

LiveJournal 是 99 年始于校园中的项目，几个人出于爱好做了这样一个应用，以实现以下功能：

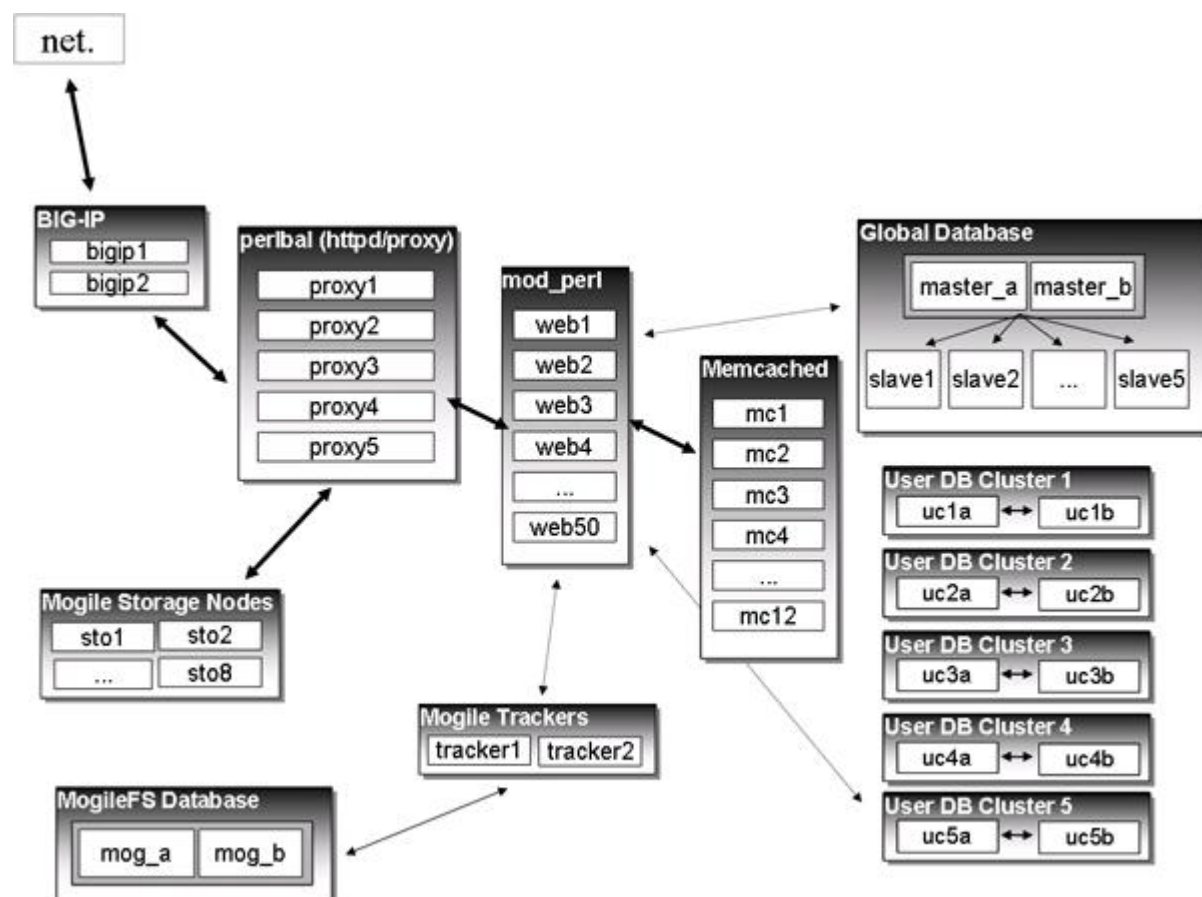
- 博客，论坛
- 社会性网络，找到朋友
- 聚合，把朋友的文章聚合在一起

LiveJournal 采用了大量的开源软件，甚至它本身也是一个开源软件。

在上线后，LiveJournal 实现了非常快速的增长：

- 2004 年 4 月份：280 万注册用户。
- 2005 年 4 月份：680 万注册用户。
- 2005 年 8 月份：790 万注册用户。
- 达到了每秒钟上千次的页面请求及处理。
- 使用了大量 MySQL 服务器。
- 使用了大量通用组件。

二、LiveJournal 架构现状概况



三、从 LiveJournal 发展中学习

LiveJournal 从 1 台服务器发展到 100 台服务器，这其中经历了无数的伤痛，但同时也摸索出了解决这些问题的方法，通过对 LiveJournal 的学习，可以让我们避免 LJ 曾经犯过的错误，并且从一开始就对系统进行良好的设计，以避免后期的痛苦。

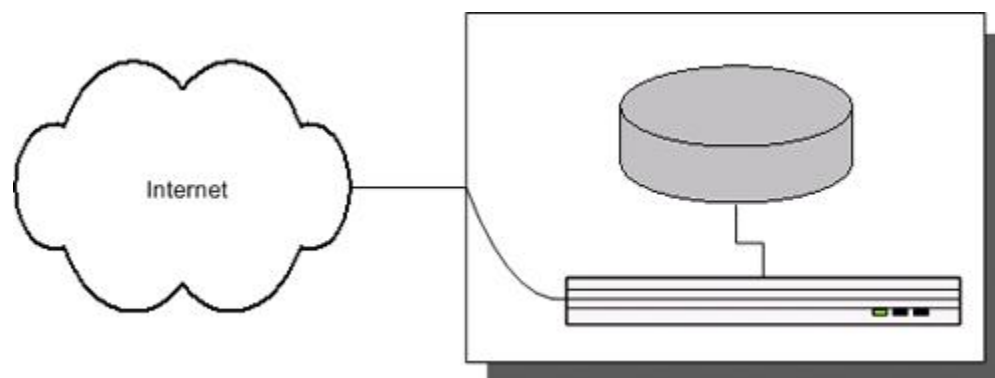
下面我们一步一步看 LJ 发展的脚步。

1、一台服务器

一台别人捐助的服务器，LJ 最初就跑在上面，就像 Google 开始时候用的破服务器一样，值得我们尊敬。这个阶段，LJ 的人以惊人的速度熟悉的 Unix 的操作管理，服务器性能出现过问题，不过还好，可以通过一些小修小改应付过去。在这个阶段里 LJ 把 CGI 升级到了 FastCGI。

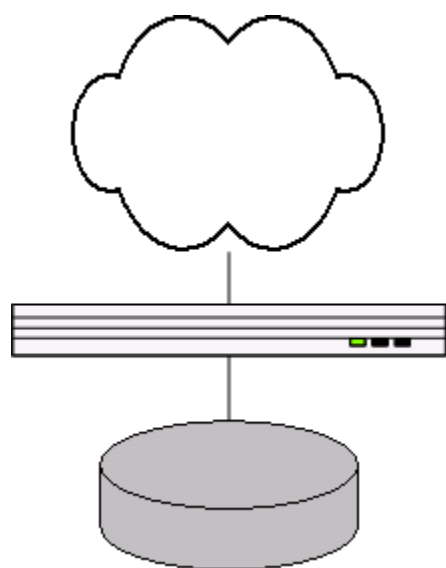
最终问题出现了，网站越来越慢，已经无法通过优化来解决的地步，需要更多的服务器，这时 LJ 开始提供付费服务，可能是想通过这些钱来购买新的服务器，以解决当时的困境。

毫无疑问，当时 LJ 存在巨大的单点问题，所有的东西都在那台服务器的铁皮盒子里装着。



2、两台服务器

用付费服务赚来的钱 LJ 买了两台服务器：一台叫做 Kenny 的 Dell 6U 机器用于提供 Web 服务，一台叫做 Cartman 的 Dell 6U 服务器用于提供数据库服务。



LJ 有了更大的磁盘，更多的计算资源。但同时网络结构还是非常简单，每台机器两块网卡，Cartman 通过内网为 Kenny 提供 MySQL 数据库服务。

暂时解决了负载的问题，新的问题又出现了：

- 原来的一个单点变成了两个单点。
- 没有冷备份或热备份。
- 网站速度慢的问题又出现了，没办法，增长太快了。

- Web 服务器上 CPU 达到上限，需要更多的 Web 服务器。

3、四台服务器

又买了两台，Kyle 和 Stan，这次都是 1U 的，都用于提供 Web 服务。目前 LJ 一共有 3 台 Web 服务器和一台数据库服务器。这时需要在 3 台 Web 服务器上进行负载均横。



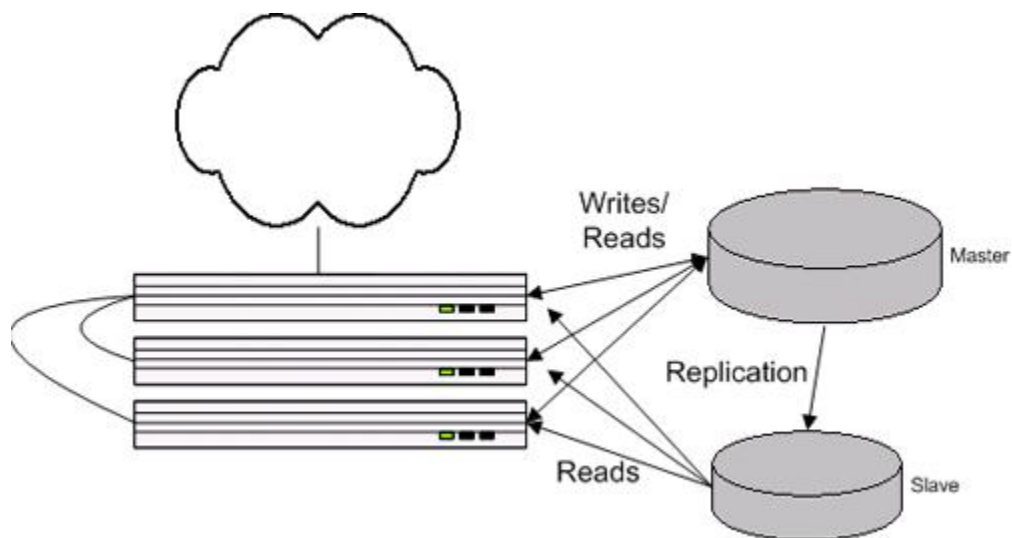
LJ 把 Kenny 用于外部的网关，使用 `mod_backend` 进行负载均横。

然后问题又出现了：

- 单点故障。数据库和用于做网关的 Web 服务器都是单点，一旦任何一台机器出现问题将导致所有服务不可用。虽然用于做网关的 Web 服务器可以通过保持心跳同步迅速切换，但还是无法解决数据库的单点，LJ 当时也没做这个。
- 网站又变慢了，这次是因为 IO 和数据库的问题，问题是怎么往应用里面添加数据库呢？

4、五台服务器

又买了一台数据库服务器。在两台数据库服务器上使用了数据库同步 (Mysql 支持的 Master-Slave 模式)，写操作全部针对主数据库 (通过 Binlog，主服务器上的写操作可以迅速同步到从服务器上)，读操作在两个数据库上同时进行 (也算是负载均横的一种吧)。

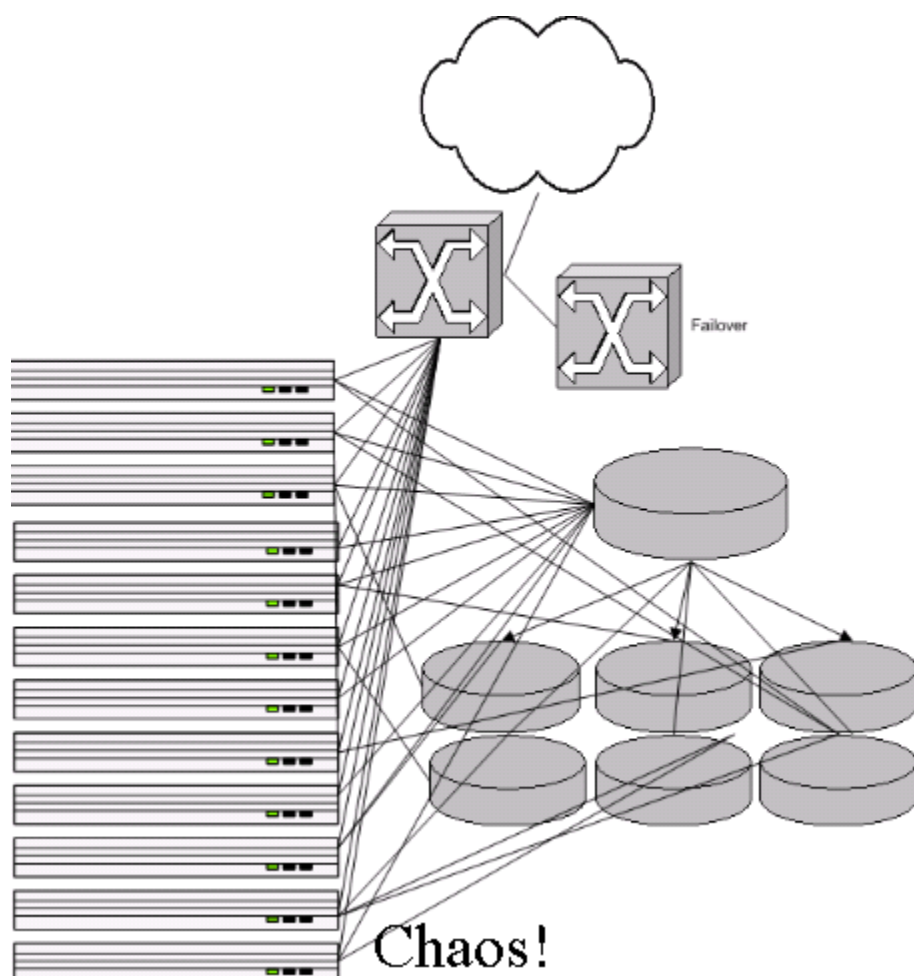


实现同步时要注意几个事项：

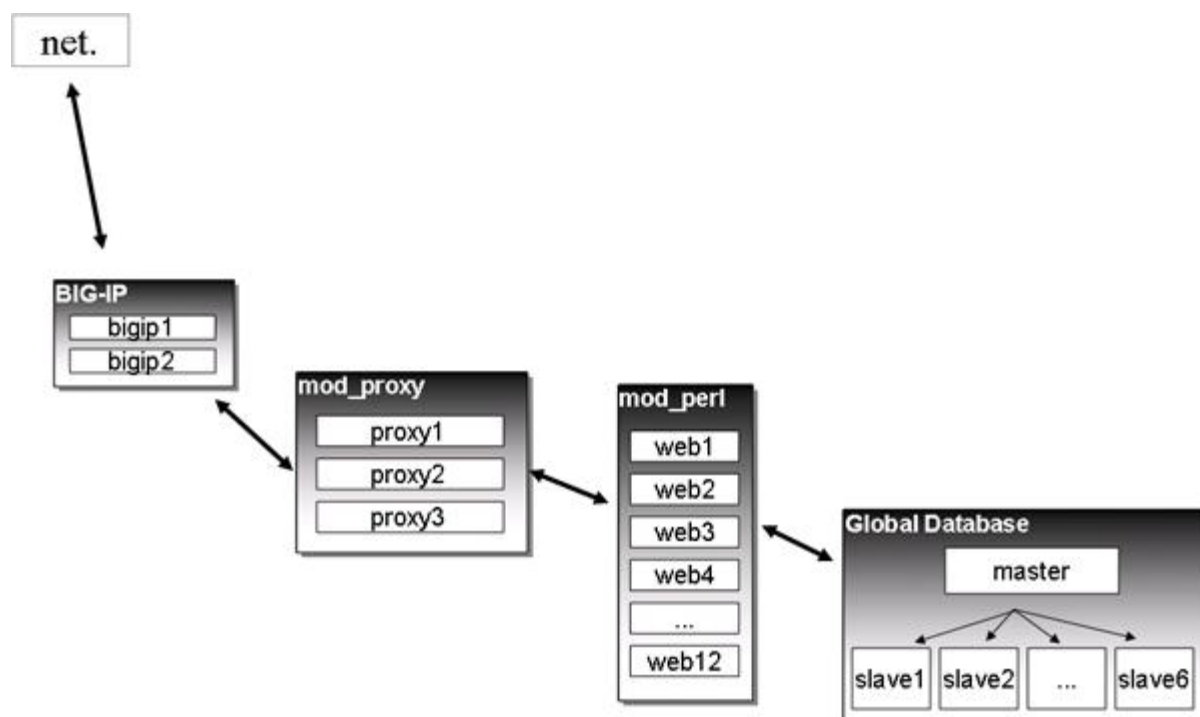
- 读操作数据库选择算法处理，要选一个当前负载轻一点的数据库。
- 在从数据库服务器上只能进行读操作
- 准备好应对同步过程中的延迟，处理不好可能会导致数据库同步的中断。只需要对写操作进行判断即可，读操作不存在同步问题。

5、更多服务器

有钱了，当然要多买些服务器。部署后快了没多久，又开始慢了。这次有更多的 Web 服务器，更多的数据库服务器，存在 IO 与 CPU 争用。于是采用了 BIG-IP 作为负载均衡解决方案。

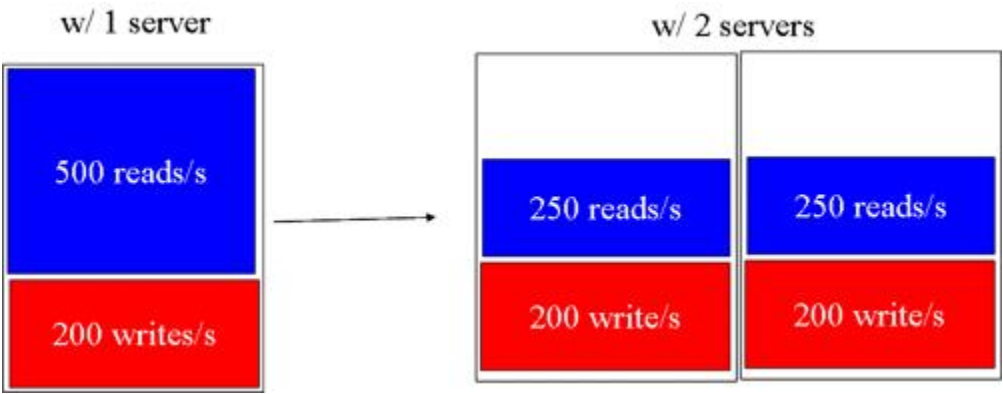


6、现在我们在哪里：

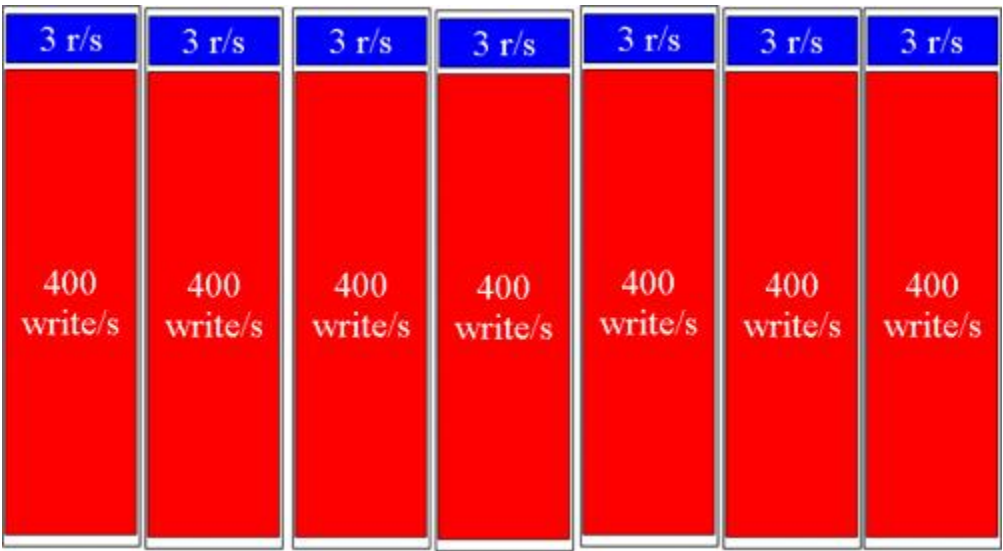


现在服务器基本上够了，但性能还是有问题，原因出在架构上。

数据库的架构是最大的问题。由于增加的数据库都是以 **Slave** 模式添加到应用内，这样唯一的好处就是将读操作分布到了多台机器，但这样带来的后果就是写操作被大量分发，每台机器都要执行，服务器越多，浪费就越大，随着写操作的增加，用于服务读操作的资源越来越少。



由一台分布到两台

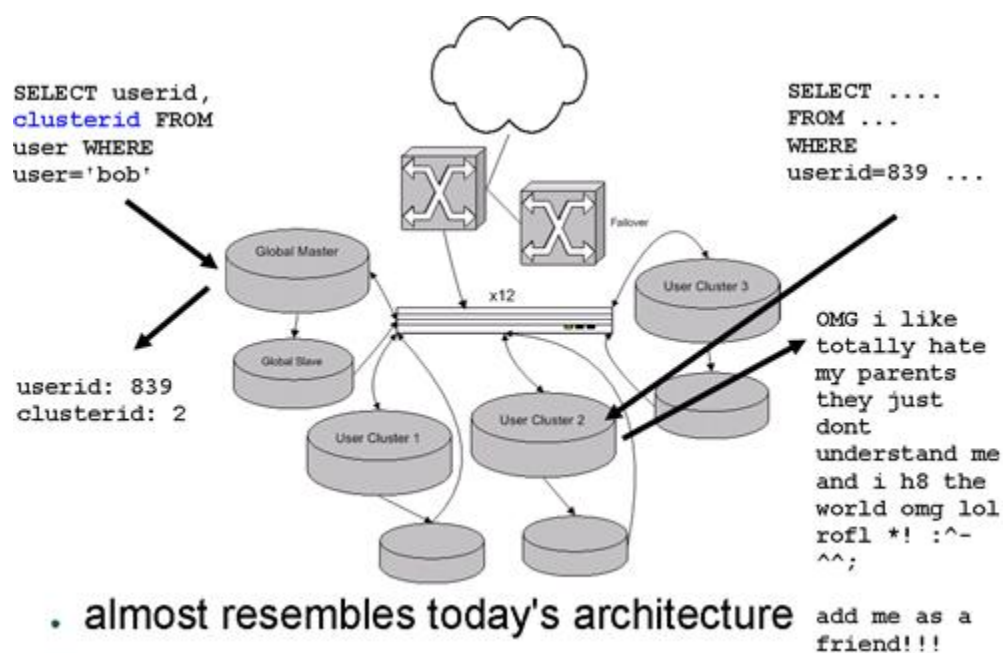


最终效果

现在我们发现，我们并不需要把这些数据在如此多的服务器上保留一份。服务器上已经做了 **RAID**，数据库也进行了备份，这么多的备份完全是对资源的浪费，属于冗余极端过度。那为什么不把数据分布存储呢？

问题发现了，开始考虑如何解决。现在要做的就是将不同用户的数据分布到不同的服务器上存储，以实现数据的分布式存储，让每台机器只为相对固定的用户服务，以实现平行的架构和良好的可扩展性。

为了实现用户分组，我们需要为每一个用户分配一个组标记，用于标记此用户的数据存放在哪一组数据库服务器中。每组数据库由一个 **master** 及几个 **slave** 组成，并且 **slave** 的数量在 2-3 台，以实现系统资源的最合理分配，既保证数据读操作分布，又避免数据过度冗余以及同步操作对系统资源的过度消耗。



由一台（一组）中心服务器提供用户分组控制。所有用户的分组信息都存储在这台机器上，所有针对用户的操作需要先查询这台机器得到用户的组号，然后再到相应的数据库组中获取数据。

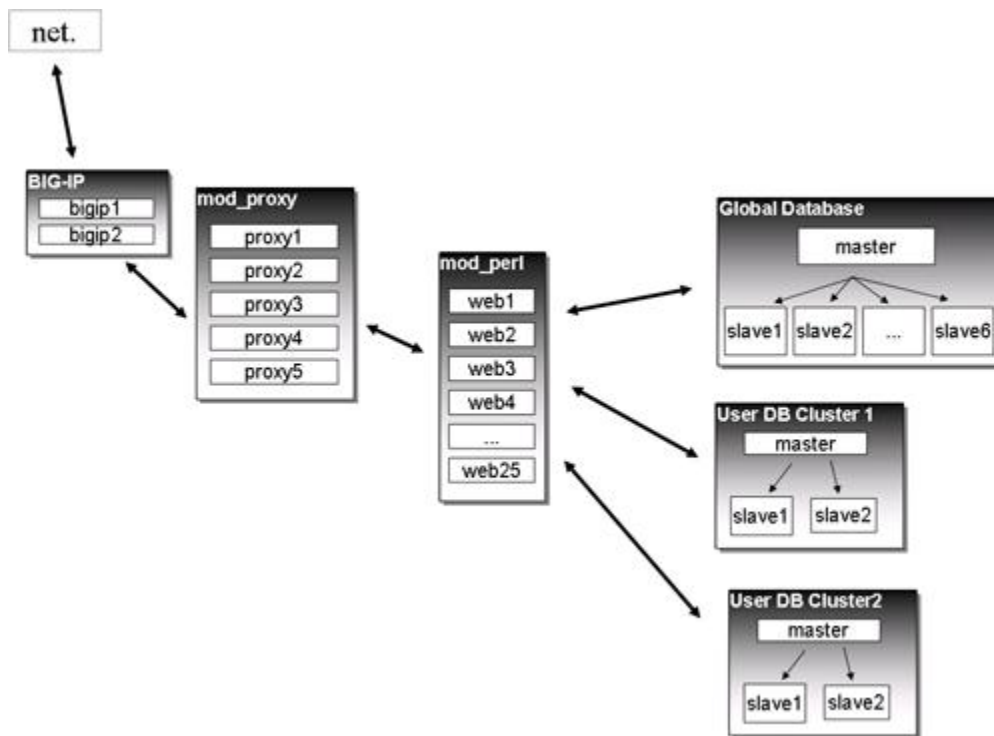
这样的用户架构与目前 LJ 的架构已经很相像了。

在具体的实现时需要注意几个问题：

- 在数据库组内不要使用自增 ID，以便于以后在数据库组之间迁移用户，以实现更合理的 I/O，磁盘空间及负载分布。
- 将 **userid**，**postid** 存储在全局服务器上，可以使用自增，数据库组中的相应值必须以全局服务器上的值为准。全局服务器上使用事务型数据库 InnoDB。

- 在数据库组之间迁移用户时要万分小心，当迁移时用户不能有写操作。

7、现在我们在哪里



问题：

- 一个全局主服务器，挂掉的话所有用户注册及写操作就挂掉。
- 每个数据库组一个主服务器，挂掉的话这组用户的写操作就挂掉。
- 数据库组从服务器挂掉的话会导致其它服务器负载过大。

对于 Master-Slave 模式的单点问题，LJ 采取了 Master-Master 模式来解决。所谓 Master-Master 实际上是人工实现的，并不是由 MySQL 直接提供的，实际上也就是两台机器同时是 Master，也同时是 Slave，互相同步。

Master-Master 实现时需要注意：

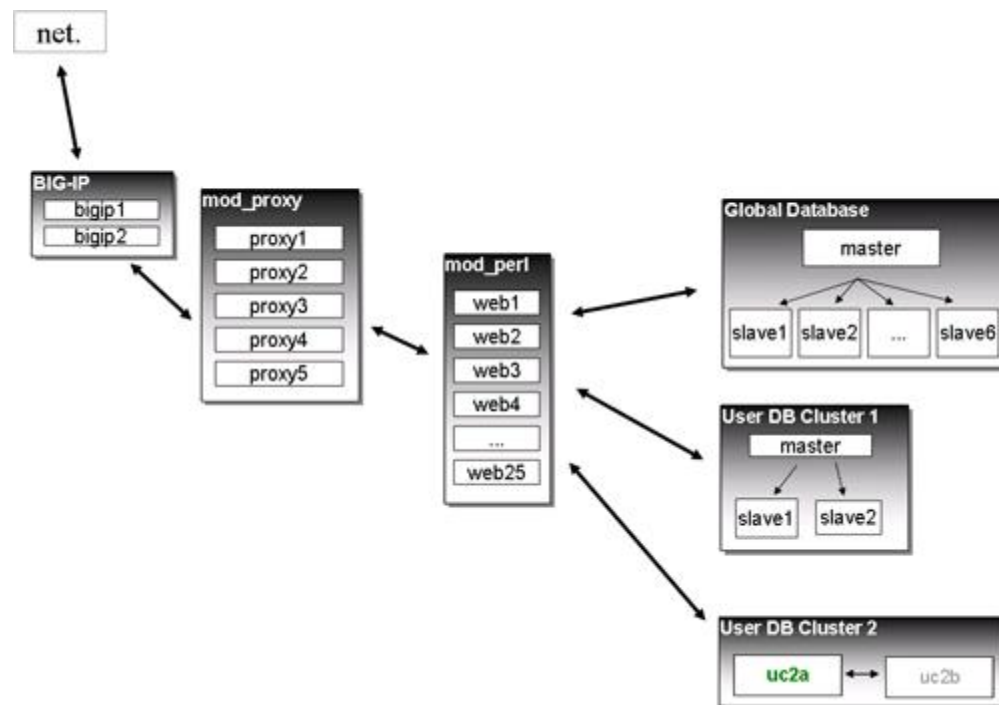
- 一个 Master 出错后恢复同步，最好由服务器自动完成。
- 数字分配，由于同时在两台机器上写，有些 ID 可能会冲突。

解决方案：

- 奇偶数分配 ID，一台机器上写奇数，一台机器上写偶数
- 通过全局服务器进行分配(LJ 采用的做法)。

Master-Master 模式还有一种用法，这种方法与前一种相比，仍然保持两台机器的同步，但只有一台机器提供服务（读和写），在每天晚上的时候进行轮换，或者出现问题的时候进行切换。

8、现在我们在哪里



现在插播一条广告，MyISAM VS InnoDB。

使用 InnoDB:

- 支持事务
- 需要做更多的配置，不过值得，可以更安全的存储数据，以及得到更快的速度。

使用 MyISAM:

- 记录日志（LJ 用它来记网络访问日志）
- 存储只读静态数据，足够快。
- 并发性很差，无法同时读写数据（添加数据可以）

- MySQL 非正常关闭或死机时会导致索引错误，需要使用 `myisamchk` 修复，而且当访问量大时出现非常频繁。

9、缓存

去年我写过一篇文章介绍 `memcached`，它就是由 LJ 的团队开发的一款缓存工具，以 `key-value` 的方式将数据存储到分布的内存中。LJ 缓存的数据：

- 12 台独立服务器（不是捐赠的）
- 28 个实例
- 30GB 总容量
- 90-93% 的命中率（用过 `squid` 的人可能知道，`squid` 内存加磁盘的命中率大概在 70-80%）

如何建立缓存策略？

想缓存所有的东西？那是不可能的，我们只需要缓存已经或者可能导致系统瓶颈的地方，最大程度的提交系统运行效率。通过对 MySQL 的日志的分析我们可以找到缓存的对象。

缓存的缺点？

- 没有完美的事物，缓存也有缺点：
- 增大开发量，需要针对缓存处理编写特殊的代码。
- 管理难度增加，需要更多人参与系统维护。
- 当然大内存也需要钱。

10、Web 访问负载均衡

在数据包级别使用 BIG-IP，但 BIG-IP 并不知道我们内部的处理机制，无法判断由哪台服务器对这些请求进行处理。反向代理并不能很好的起到作用，不是已经够快了，就是达不到我们想要的效果。

所以，LJ 又开发了 `Perlbal`。特点：

- 快，小，可管理的 `http web` 服务器/代理

- 可以在内部进行转发
- 使用 Perl 开发
- 单线程，异步，基于事件，使用 `epoll` , `kqueue`
- 支持 Console 管理与 http 远程管理，支持动态配置加载
- 多种模式：web 服务器，反向代理，插件
- 支持插件：GIF/PNG 互换？

11、MogileFS

LJ 使用开源的 **MogileFS** 作为分布式文件存储系统。MogileFS 使用非常简单，它的主要设计思想是：

- 文件属于类（类是最小的复制单位）
- 跟踪文件存储位置
- 在不同主机上存储
- 使用 MySQL 集群统一存储分布信息
- 大容易廉价磁盘

到目前为止就这么多，更多文档可以在 <http://www.danga.com/words/> 找到。[Danga.com](http://www.danga.com/) 和 [LiveJournal.com](http://www.livejournal.com/) 的同学们拿这个文档参加了两次 MySQL Con，两次 OS Con，以及众多的其它会议，无私的把他们的经验分享出来，值得我们学习。在 web2.0 时代快速开发得到大家越来越多的重视，但良好的设计仍是每一个应用的基础，希望 web2.0 们在成长为 Top500 网站的路上，不要因为架构阻碍了网站的发展。

参考资料：http://www.danga.com/words/2005_oscon/oscon-2005.pdf

I Craigslist 的数据库架构

作者：Fenng | [English Version](#) 【可以转载，转载时务必以超链接形式标明文章原始出处和作者信息及版权声明】

网址：http://www.dbanotes.net/database/craigslist_database_arch.html

(插播一则新闻：竞拍这本《Don't Make Me Think》，我出价 RMB 85，留言的不算--不会有恶意竞拍的吧？要 Ping 过去才可以，失败一次，再来)

Craigslist 绝对是互联网的一个传奇公司。根据以前的一则报道：

每月超过 1000 万人使用该站服务，月浏览量超过 30 亿次，(Craigslist 每月新增的帖子近 10 亿条??)网站的网页数量在以每年近百倍的速度增长。Craigslist 至今却只有 18 名员工(现在可能会多一些了)。

Tim O'reilly 采访了 Craigslist 的 Eric Scheide，于是通过这篇 [Database War Stories #5: craigslist](#) 我们能了解一下 Craigslist 的数据库架构以及数据量信息。

数据库软件使用 MySQL。为充分发挥 MySQL 的能力，数据库都使用 64 位 Linux 服务器, 14 块本地磁盘(72*14=1T ?), 16G 内存。

不同的服务使用不同方式的数据库集群。

论坛

1 主(master) 1 从(slave)。Slave 大多用于备份。mysam 表。索引达到 17G。最大的表接近 4200 万行。

分类信息

1 主 12 从。Slave 各有各的用途。当前数据包括索引有 114 G，最大表有 5600 万行(该表数据会定期归档)。使用 mysam。分类信息量有多大？"Craigslist 每月新增的帖子近 10 亿条"，这句话似乎似乎有些夸张，Eric Scheide 说昨日就超过 330000 条数据，如果这样估计的话，每个月的新帖子信息大约在 1 亿多一些。

归档数据库

1 主 1 从。放置所有超过 3 个月的帖子。与分类信息库结构相似但是更大，数据有 238G，最大表有 9600 万行。大量使用 Merge 表，便于管理。

搜索数据库

4 个集群用了 16 台服务器。活动的帖子根据地区/种类划分，并使用 mysam 全文索引，每个只包含一个子集数据。该索引方案目前还能撑住，未来几年恐怕就不行了。

Authdb

1 主 1 从，很小。

目前 Craigslist 在 Alexa 上的排名是 30，上面的数据只是反映采访当时(April 28, 2006)的情况，毕竟，Craigslist 数据量还在每年 200% 的速度增长。

Craigslist 采用的数据解决方案从软硬件上来看还是低成本的。优秀的 MySQL 数据库管理员对于 Web 2.0 项目是一个关键因素。

--EOF--

I Second Life 的数据拾零

作者: Fenng | [English Version](#) 【可以转载，转载时务必以超链接形式标明文章原始出处和作者信息及版权声明】

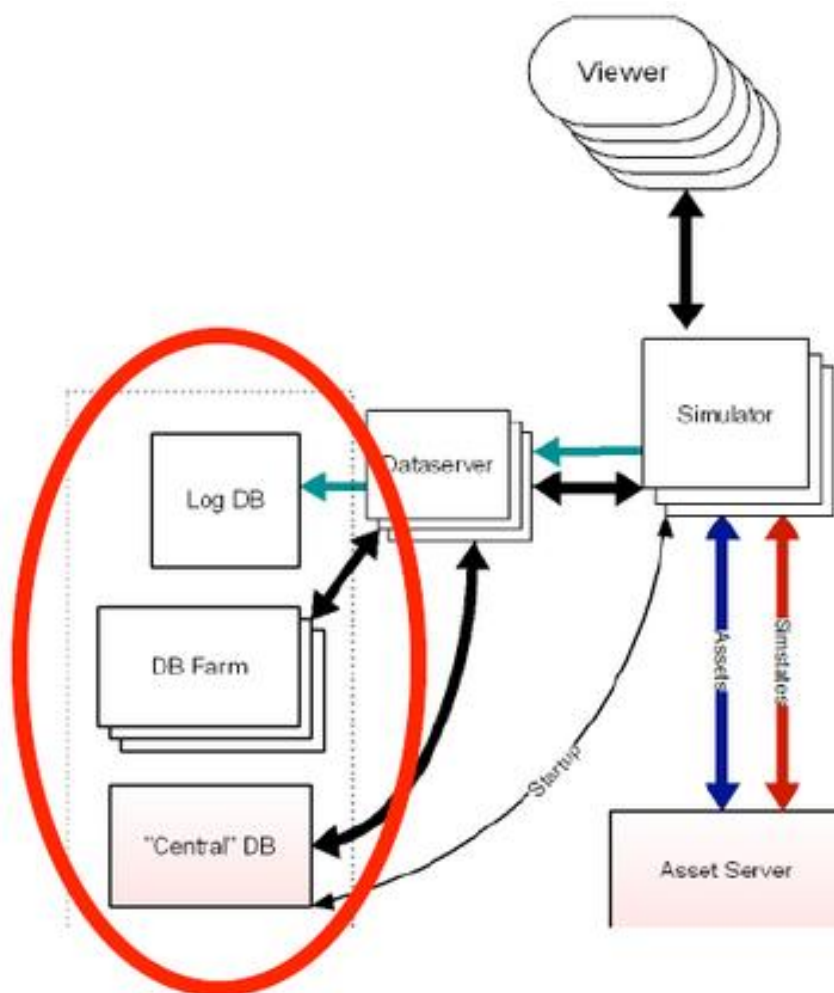
网址: http://www.dbanotes.net/review/second_life.html

Matrix 似乎提前来到我们身边。从 06 年开始，陆续看到多次关于 Second Life(SL) 的报道。因为自己的笔记本跑不起来 SL 的客户端，所以一直没有能体会这个虚拟世界的魅力。今天花了一点时间，读了几篇相关的文档。

RealNetworks 前 CTO Philip Rosedale 通过 [Linden 实验室](#) 创建了 Second Life, 2002 年这个项目开始 Alpha 版测试, 当时叫做 LindenWorld。

2007 年 2 月 24 日号称已经达到 400 万用户(用户在游戏中被称为 "Residents", 居民)。2001 年 2 月 1 日, 并发用户达到 3 万。并发用户每月的增长是 20%。这个 20% 现在看起来有些保守了, 随着媒体的关注, 增长的会有明显的变化。系统的设计目标是 10 万并发用户, 系统的复杂度不小, 但 Linden 实验室对 SL 的可扩展能力信心满满。

目前在旧金山与达拉斯共有 2000 多台(现在恐怕 3000 也不止了吧) Intel/AMD 服务器来支撑整个虚拟世界(refer [here](#))。64 位的 AMD 服务器居多。操作系统选用的 Debian Linux, 数据库是 MySQL。通过 Tim O'relly 的这篇 [Web 2.0 and Databases Part 1: Second Life](#), 可以了解到一点关于 SL 数据库建设的信息。在 Second Life 中每个地理区域都是运行在服务器软件单一实例上的, 叫做"模拟器"或者简称是 "sim", 每个 Sim 负责 16 英亩的虚拟土地。当用户在相邻的 Sim 间移动, 实际上是从一个处理器(或是服务器)移动到另一个。根据这篇[访谈](#), 用户当前所在 Sim 的信息, 以及用户本身的账户信息是存储在一个中心数据库上的。



SL 的客户端软件的下载使用了 Amazon 的 [S3](#) 服务。

一点感想: MySQL 真是这波 Web 2.0 大潮中最大赢家之一啊

--EOF--

I 原 eBay 架构的思想金矿

英文来源: <http://www.manageability.org/blog/stuff/about-ebays-architecture>

杨争 /译

了解一件事情是怎么做的一个正确的方式是看看它在现实中是怎么做的。软件工业一直以来都在为"很多 idea 仅仅在理论上说说"所困惑。与此同时, 软件厂商不断地把这些 idea 作为最佳实践推销给大家。

很少的软件开发者亲眼目睹过大规模可扩展的架构这一领域。幸运的是, 有时我们可以看到和听到关于这方面公开发表的资料。我读过一些好的资料关于 google 的硬件基础设施的设计以及 yahoo 的页面渲染专利。现在, 另一个互连网的巨人, eBay, 给我们提供了其架构的一些资料(译者注: 指的是"一天十亿次的访问—采用 Core J2EE Pattern 架构的 J2EE 系统"这篇文章)。

这篇文章提供了很多信息。然而, 我们将只对那些独特的和我感兴趣的那部分进行评论。

给我留下深刻印象是 eBay 站点的 99.92%的可用性和 380M page 的页面数据。除此之外, 每周近 3 万行代码的改动, 清楚明白地告诉我们 ebay 的 java 代码的高度扩展性。

eBay 使用 J2EE 技术是如何做到这些的。eBay 可扩展性的部分如下:

Judicious use of server-side state

No server affinity

Functional server pools

Horizontal and vertical database partitioning

eBay 取得数据访问的线性扩展的做法是非常让人感兴趣的。他们提到使用"定制的 O-R mapping" 来支持本地 Cache 和全局 Cache、lazy loading, fetch sets (deep and shallow) 以及读取和提交更新的子集。而且, 他们只使用 bean 管理的事务以及使用数据库的自动提交和 O-R mapping 来 route 不同的数据源。

有几个事情是非常令人吃惊的。第一, 完全不使用 Entity Beans, 只使用他自己的 O-R mapping 工具(Hibernate anyone?)。第二、基于 Use-Case 的应用服务器划分。第三、数据库的划分也是基于 Use-Case。最后是系统的无状态本性以及明显不使用集群技术。

下面是关于服务器状态的引用:

基本上我们没有真正地使用 server-side state。我们可能使用它, 但现在我们并没有找到使用它的理由。....。如果需要状态化的话, 我们把状态放到数据库中;需要的时候我们再从数据库中取。我们不必使用集群。也就不为集群做任何工作。

总之, 你自己不必为架构一台有状态的服务器所困扰, 更进一步, 忘掉集群, 你不需要它。现在看看功能划分:

我们有一组或者一批机器, 上面运行的应用是某个具体的 use case, 比如搜索功能有他们自己的服务器群, 我们可以采用不同的调优策略, 原因是浏览商品这个基本上是只读的用例和卖一件商品这个读写的用例在执行的时候是不同。在过去四五年我们一直采用水平数据库划分达到我们需要的可用性和线性扩展性。

总之，不要把你的应用和数据库放在一个 **giant machine**，仅仅使用 **servers pools**，每个 **pools** 对应一个 **Use Case**。听起来是否类似 **Google** 的策略。

下面是关于水平划分的一些介绍：

基于内容的路由可以实现系统的水平线性扩展。所以，想象一下，如果 **eBay** 某天拥有 **6000** 万种商品，我们不必把这些数据存储到一台超级 **Sun** 服务器上。.....也许我们可以把这些数据库放到许多台 **Sun** 服务器，但是我们怎么取到我们需要的数据呢？**eBay** 提出了基于内容路由的方法。这种方法通过一定的规则，从 **20** 台物理服务器中找到我需要的数据。更 **cool** 的事情是这里还定义了 **failover** 的策略。

最后，下面一句话描述了未来采用更加松散耦合的架构：

使用消息系统来耦合不同的 **Use Case** 是我们研究的内容。

是不是觉得很奇怪，最初这篇文章是介绍 **J2EE** 设计模式的？关键的线性扩展的思想几乎和 **Patterns** 无关。是的，**eBay** 采用设计模式组织他们的代码。然而过分强调设计模式将失去对整体的把握。**eBay** 架构关键的思想是无状态的设计，使用灵活的，高度优化的 **OR-mapping** 层以及服务器基于 **use cases** 划分。设计模式是好的，然而不能期望它使应用具有线性扩展性。

总之，**eBay** 和 **Google** 的例子表明以 **Use-Case** 为基础组成的服务器 **pools** 的架构比几个大型计算机证明是具有更好线性扩展性的和可用性。当然，厂商害怕听到这样的结论。然而，部署这么多服务器的最大麻烦是如何管理好他们。-)

我的总结：

eBay 采用设计模式达到 **eBay** 架构的分层，各层（表示层、商业逻辑层、数据访问层）之间松散耦合，职责明确，分层提高了代码的扩展性和程序开发的效率。

eBay 采用无状态的设计，灵活的、高度优化的 **OR-mapping** 层以及服务器基于 **use cases** 划分，达到应用之间的松散耦合，提高系统的线性扩展性。

为什么要求系统具有可线性扩展，目的就是当网站的访问量上升的时候，我们可以不用改动系统的任何代码，仅仅通过增加服务器就可以提高整个网站的支撑量。

I 原一天十亿次的访问—eBay 架构（一）

版权声明：如有转载请求，请注明出处：<http://blog.csdn.net/yzhz>

本文来自于 2003JavaOne (<http://java.sun.com/javaone/>) 上的一篇文章。我把它翻译成中文，有些不重要的部分我已略去。虽然是 2003 年的文章，但其中的 **J2EE** 设计方案还是值得我们去学习的，而且这个架构本身就是面向未来的。

eBay 作为全球最大的网络交易市场赢得了市场的尊重，作为技术人员我们对其后台架构如何能够支撑起这个庞然大物都会感兴趣。每天十亿次访问量，6900 万注册会员，1600 万商品这些天文般的数字意味着它每天承受着巨大的并发访问量，而且 **eBay** 上大量页面都不是静态页面。

这篇介绍 **eBay** 架构的文章一定能对我们的项目设计和开发起到很好的指导作用。

eBay 的架构是 **eBay** 的工程师和 **Sun** 的工程师共同设计完成的。

下面文章中斜体字是我的注释或者感想，其他的都是原文翻译。

作者：Deepak Alur、Arnold Goldberg、Raj Krishnamurthy

翻译：杨争

一天十亿次的访问

采用 Core J2EE Pattern 架构的 J2EE 系统

详细了解 Core J2EE Pattern 可以查看此链接

<http://java.sun.com/blueprints/corej2eepatterns/Patterns/>

目标：

通过本文，学习如何采用 Core J2EE Patterns 架构具有高度扩展性多层的 J2EE 应用。

作者：

Deepak Alur

- Senior Software Architect, SunPS program
- Co-author of Core J2EE Patterns
- Sun-eBay V3 Architecture—Team leader

Arnold Goldberg

- Lead Architect—eBay.com Platform
- Led V3 architecture, design and implementation

Raj Krishnamurthy

- Software Architect, SunPS program
- Sun-eBay V3 Architecture team—Key member

议程：

入门和 Core J2EE Patterns

eBay.com 三层架构的目标

关键架构和技术决策

eBay.com 如何应用 Core J2EE Patterns

结论

一、入门和 Core J2EE Patterns

1、目标：

- eBay.com 网站的架构
- 架构中模式的地位
- 使用 Core J2EE Patterns 的好处

2、eBay 介绍

(1) 使命

- 1、全球交易平台
- 2、拍卖、定价、B2C、B2B

(2) 统计数据

- 6900 万注册会员
- 28000 个分类，1600 万商品
- 2002 年营业额：148 亿 7 千万美元
- 全球社区
- 每天十亿次访问量
- 1200 多个 URL

3、eBay 旧的二层架构及其存在的问题

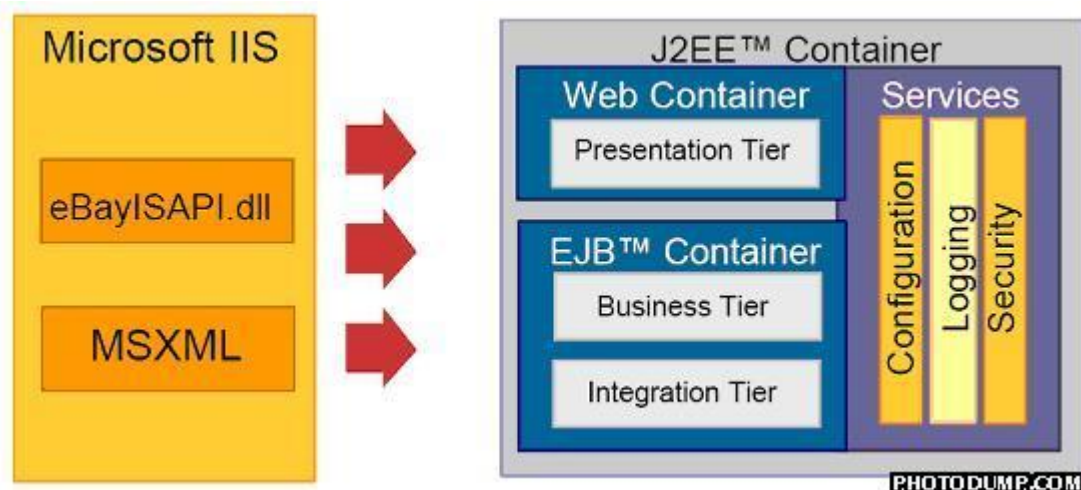
(1) ebay 旧的二层架构

- 集成在一起的两层架构（架构中各组件之间的耦合度高）
- 330 万行 C++ ISAPI DLL
- 面向功能的设计
- Not for systemic qualities

(2) 二层架构存在的问题

- 阻碍商业创新（可扩展性不够）
- 随着访问量增大，系统线性扩展性面临着挑战（无法通过仅仅增加硬件投入，扩充系统的支撑量）
- 高额的维护成本
- 不便于“重构”（代码很难通过重构来改善）
- Architects in constant Fire-Fighting Mode

4、2000 年底开始三层架构改造



系统向分层、松散耦合、模块化、基于标准的架构过渡

版权声明：如有转载请求，请注明出处：<http://blog.csdn.net/yzhz>

5、eBay 架构的改造是基于下面这本书介绍的模式

core J2EE Pattern 最佳实践和设计策略第二版，sun 官方网站也提供 core J2EE Pattern，见 <http://java.sun.com/blueprints/corej2eepatterns/Patterns/>

该书介绍了 21 种 J2EE 设计模式，我们可以把他们归类到三层中。

(1)、表示层的设计模式：

- Intercepting Filter (X)
- Front Controller (X)
- Application Controller (X)
- Context Object (X)
- View Helper
- Composite View
- Service To Worker (X)
- Dispatcher View

带(X)表示这些设计模式在 eBay.com 的架构中采用了。

(2)、商业逻辑层的设计模式：

- Business Delegate
- Service Locator (X)
- Session Facade
- Application Service (X)
- Business Object (X)
- Composite Entity
- Transfer Object (X)
- Transfer Object Assembler (X)
- Value List Handler (X)

带(X)表示这些设计模式在 eBay.com 的架构中采用了。

3、集成层（也称为数据访问层）设计模式：

- Data Access Object (X)
- Service Activator
- Domain Store (X)
- Web Service Broker (X)

带(X)表示这些设计模式在 eBay.com 的架构中采用了。

二、eBay 三层架构的目标

1、目标

高可用性、高可靠性、可线性扩展，建立实现系统的无缝增长。

高开发效率，支持新功能的快速交付。

可适应未来的架构，应变将来商业的更新需求。

eBay 的系统可用性 2002 年已到了 99.92%。(令人叹服)，每季度网站新增十五个重大功能，

每个星期将近 3 万行代码在修改，3 个星期内可以提供一个国际化版本。

2、为了可适应未来的架构，ebay 采用了下面的做法

采用 J2EE 模式

Only adopt Technology when required

Create new Technology as needed

大量的性能测试

大量的容量计划

大量关键点的调优

Highly redundant operational infrastructure and the technology to leverage it

3、为了实现可线性扩展，ebay 采用了下面的做法：

- (1) 合理地使用 server state
- (2) No server affinity
- (3) Functional server pools。
- (4) Horizontal and vertical database partitioning。

ebay 架构采用了服务器分块化的概念，每台服务器上的应用与它的 use case 有关，即 server pool 中的一部分服务器专门用于登陆，一部分服务器专门用于显示商品信息。毕竟不同 use case 访问数据库的方式不同，比如“显示商品信息” use case 只是只读操作。而且由于是只读操作，数据库的压力会比较低，我可以只采用几台服务器来承担这部分操作，而更多的服务器用于读写操作多的 use case，这样合理地使用服务器资源。

由于不同的应用放在不同的服务器上，这里就涉及到用户状态的复制问题。这就是第一条 ebay 要求合理地使用 server state 的原因，就我所知，ebay 的用户状态只有很少保存在 session 中，ebay 把用户的状态放到了数据库和 cookie 中。

4、为了使得数据访问可线性扩展

- (1) 建模我们的数据访问层
- (2) 支持 Support well-defined data access patterns

Relationships and traversals

本地 cache 和全局 cache

- (3) 定制的 O-R mapping—域存储模式
- (4) Code generation of persistent objects
- (5) 支持 lazy loading
- (6) 支持 fetch sets (shallow/deep fetches)
- (7) 支持 retrieval and submit (Read/Write sets)

5、为了使数据存储可线性扩展，eBay 采用了下列做法

(1) 商业逻辑层的事务控制

只采用 Bean 管理的事务

Judicious use of XA

数据库的自动提交

(2) 基于内容的路由

运行期间采用 O-R Mapping，找到正确的数据源

支持数据库的水平线性扩展

Failover hosts can be defined

(3) 数据源管理

动态的

Overt and heuristic control of availability

如果数据库宕机，应用可以为其他请求服务。

6、应变未来采用的技术

(1) 消息系统

子系统之间、数据库之间松散耦合

J2EE 的 Message Driven Beans

(2) SOAP

对于外部开发者和合作伙伴，通过可用的工具和最佳实践来平衡我们的平台

采用 SOAP 来标准化不同 eBay 应用之间进程内部的通信

采用 SOAP 满足我们的 QoS 需求

四、将 J2EE 的设计模式应用到 eBay 中

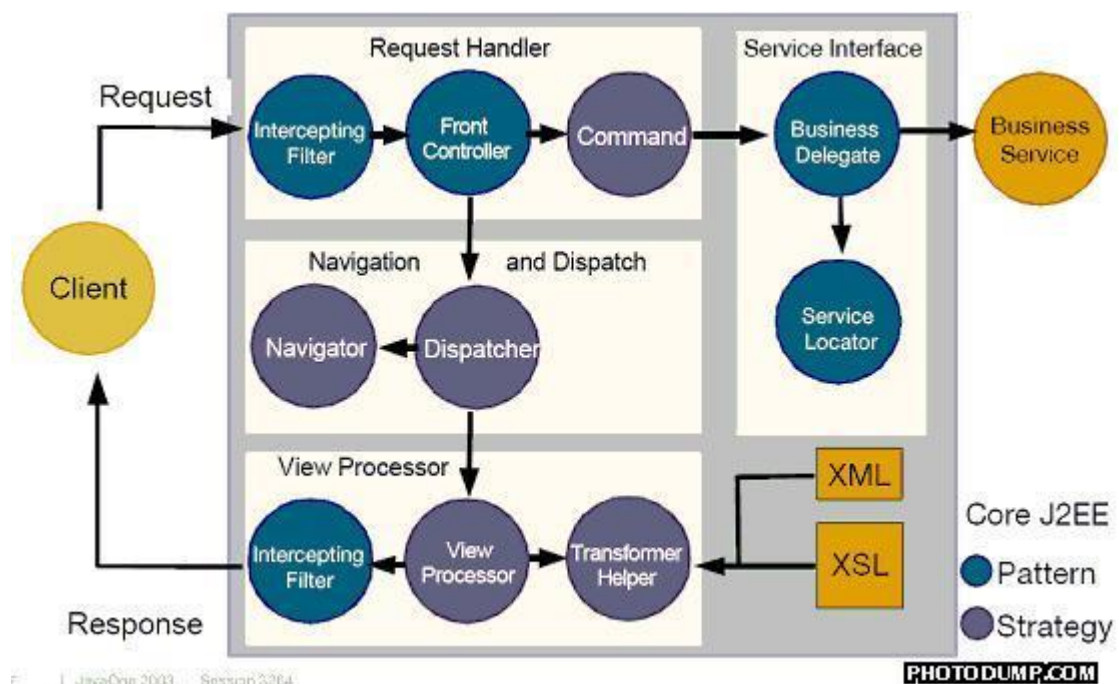
介绍了三个 Use cases 例子，“查看账号”，“查看商品”，“eBayAPI”，介绍了这三个 use case 如何采用 J2EE 的设计模式实现其设计。（略去）

原 一天十亿次的访问—eBay 架构（三）

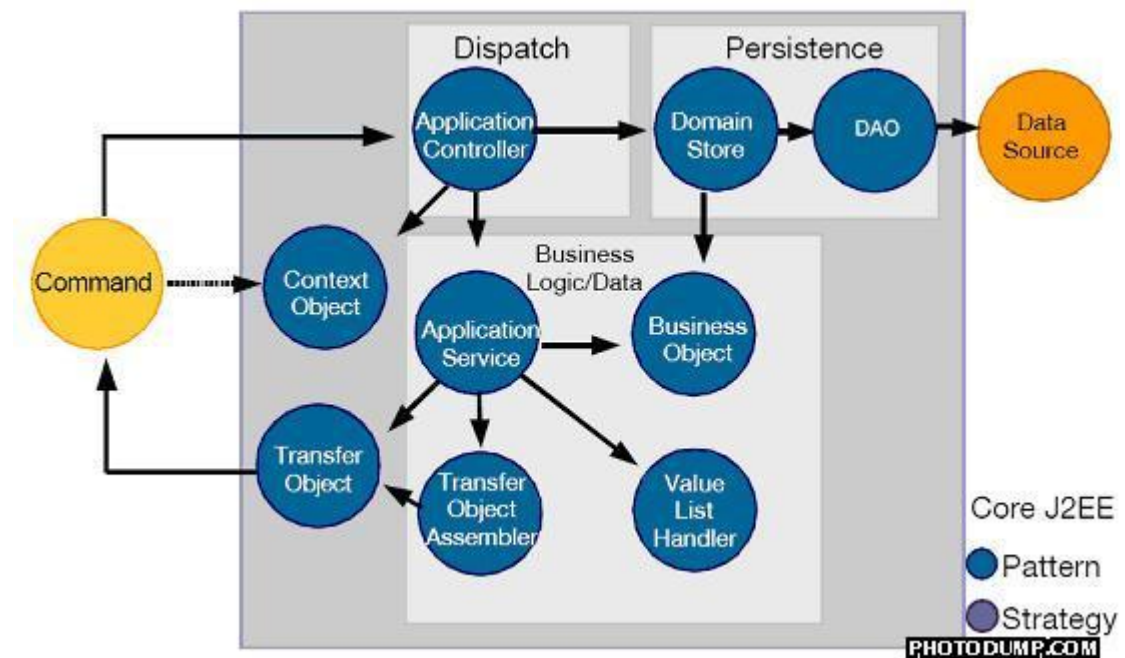
版权声明：如有转载请求，请注明出处：<http://blog.csdn.net/yzhz>

五、结论

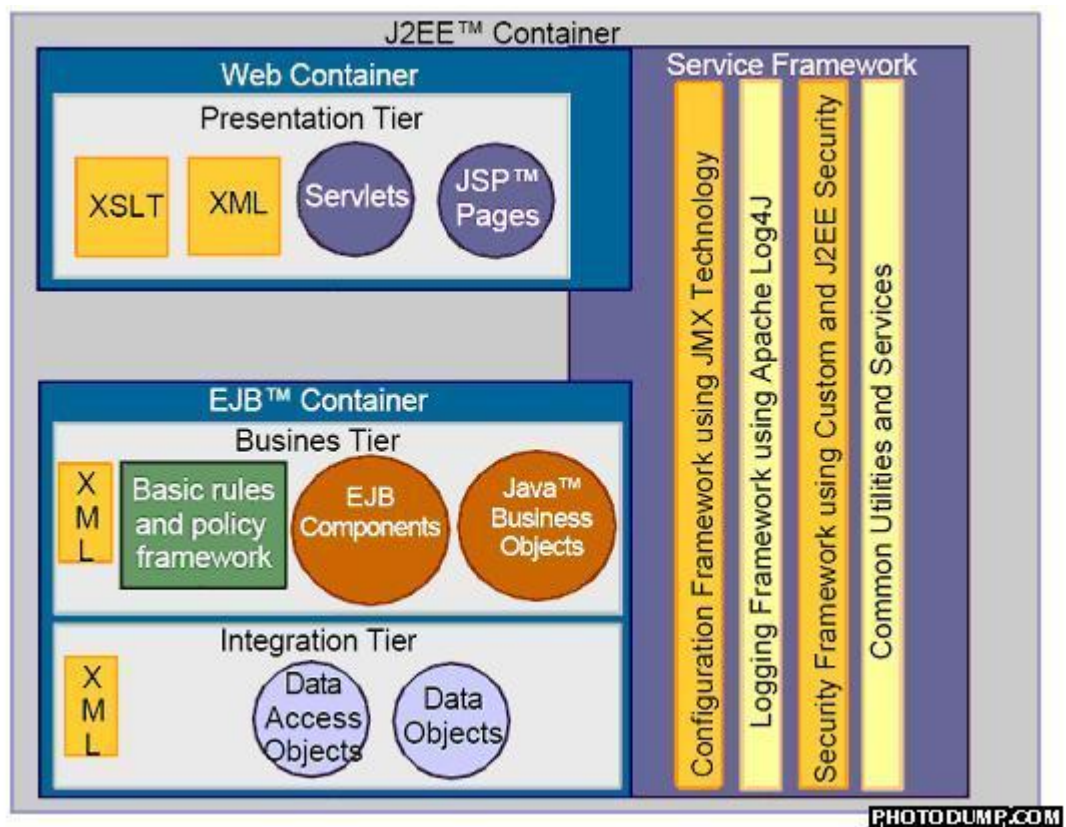
1、表示层架构



2、商业逻辑层架构



3、eBay 整体架构



4、总结

(1) eBay.com 的架构采用了 J2EE 核心模式

-使你不用重新发明轮子，提高系统重用性

-经过实践证明的解决方案和策略

-J2EE 核心模式可以成为 Developer 和 Architect 的词汇

-更快的开发效率

(2) 在你开发项目中学习和采用这些设计模式

(3) 参与到模式的社区中。

5、看了这么多，如果你能记得些什么的话，希望是下面这段话：

模式在开发和设计中是非常有用的。模式能帮助你达到设计的重用、加快开发进度、降低维护成本，提高系统和代码的可理解性。

我的体会：

1、eBay 架构的主体是采用 J2EE 的核心设计模式设计的，我们在实际项目中可根据我们应用的需求采用适合我们应用的设计模式。毕竟我们看到 eBay 的架构也不是用了 J2EE 核心设计模式中提到的所有模式，而是根据项目的实际情况采用了部分适合其本身的模式。

2、需要澄清的是：这些设计模式是 J2EE 的设计模式，而不是 EJB 的设计模式。如果你的架构没有采用 EJB，你仍然可以使用这些设计模式。

3、本文中除了介绍如何采用 J2EE 核心模式架构 eBay 网站，还介绍了 eBay 架构为了支持线性扩展而采用的一些做法，我觉得这些做法很有特点，不仅可以大大提高系统的线性扩展性，而且也能大大提高网站的性能。这些我会有另外一篇文章介绍给大家。

I 七种缓存使用武器 为网站应用和访问加速发布时间：

Web 应用中缓存的七种武器：

1 数据库的缓存

通常数据库都支持对查询结果的缓存，并且有复杂的机制保证缓存的有效性。对于 MySQL, Oracle 这样的数据库，通过合理配置缓存对系统性能带来的提升是相当显著的。

2 数据连接驱动的缓存。

诸如 PHP 的 ADODB，J2EE 的连接驱动，甚至如果把 Hibernate 等 ORM 也看成连接器的话。这里的缓存有效机制就不是那么强了，使用此步的方法实现缓存的一个最好的优点就是我们取数据的方式可以保持不变。例如，我调用

`$db->CacheGetAll("select * from table");` 的语句不需要改变，可以透明实现缓存。这主要应用于一些变化不大的数据上，例如一些数据字典是不经常变化的。

3 系统级的缓存

可以在系统内通过 **Cache** 库，自行对需要的数据进行缓存，例如一个树桩菜单生成十分消耗资源，那可以将这个生成的树缓存起来。这样做的缺点是，当这颗树的某些地方被更新时，你需要手动更新缓存内的东西。使用的缓存库都可以有不同的缓存方法，有的把内容放在硬盘上，有的放在内存里面，如果你把内容模拟成硬盘来缓存，速度当然也能提升不少。

4 页面级的缓存

这个在内容管理系统里面用的最多。也就是生成静态页面。这里面缓存控制机制最为复杂，一般也没有什么包治百病的方法，只有具体情况具体分析。通常生成的静态页面你需要有一个机制去删除过时的，或访问很少的页面，以保证检索静态页面的速度。

5 使用预编译页面和加载为 **FastCGI** 的办法

对于 **PHP**，可以使用 **zend** 等编译引擎，对于 **JSP** 本身就是预编译。而 **FastCGI** 的原理就是将脚本预先加载起来，不用每次执行都去读，这和 **JSP** 预编成 **Servlet**，然后加载的道理是一样的。

6 前置缓存

可以使用 **Squid** 作为 **Web** 服务器的前置缓存。

7 做集群

对数据库作集群，对 **web** 服务器作集群，对 **Squid** 前置机做集群

对于新手来说，如果你的程序要是恰死，首先你要检查代码是否有错误，是否存在内存泄漏，如果都没有，那么通常问题出在数据库连接上面。

综合应用上面的缓存方法，开发高负载的 **Web** 应用成就很容易了。

I 可缓存的 **CMS** 系统设计

2007-06-03 13:41:16 作者: chedong 来源: www.chedong.com 标签: cms cache 设计 (English)

文章转载自互联网，如果您觉得我们侵权了，请联系[管理员](#)，我们会立刻处理。

对于一个日访问量达到百万级的网站来说，速度很快就成为一个瓶颈。除了优化内容发布系统的应用本身外，如果能把不需要实时更新的动态页面的输出结果转化成静态网页来发布，速度上的提升效果将是显著的，因为一个动态页面的速度往往会比静态页面慢 2—10 倍，而静态网页的内容如果能被缓存在内存里，访问速度甚至会比原有动态网页有 2—3 个数量级的提高。

- [动态缓存和静态缓存的比较](#)
- [基于反向代理加速的站点规划](#)
- [基于 apache mod_proxy 的反向代理加速实现](#)
- [基于 squid 的反向代理加速实现](#)

- 面向缓存的页面设计
- 应用的缓存兼容性设计：

HTTP_HOST/SERVER_NAME 和 REMOTE_ADDR/REMOTE_HOST 需要用 HTTP_X_FORWARDED_HOST/HTTP_X_FORWARDED_SERVER 代替

后台的内容管理系统的页面输出遵守可缓存的设计，这样就可以把性能问题交给前台的缓存服务器来解决了，从而大大简化 CMS 系统本身的复杂程度。

静态缓存和动态缓存的比较

静态页面的缓存可能有 2 种形式：其实主要区别就是 CMS 是否自己负责关联内容的缓存更新管理。

1. 静态缓存：是在新内容发布的同时就立刻生成相应内容的静态页面，比如：2003 年 3 月 22 日，管理员通过后台内容管理界面录入一篇文章后，就立刻生成 <http://www.chedong.com/tech/2003/03/22/001.html> 这个静态页面，并同步更新相关索引页上的链接。
2. 动态缓存：是在新内容发布以后，并不预先生成相应的静态页面，直到对相应内容发出请求时，如果前台缓存服务器找不到相应缓存，就向后台内容管理服务器发出请求，后台系统会生成相应内容的静态页面，用户第一次访问页面时可能会慢一点，但是以后就是直接访问缓存了。

如果去 ZDNet 等国外网站会发现他们使用的基于 *Vignette* 内容管理系统都有这样的页面名称：0,22342566,300458.html。其实这里的 0,22342566,300458 就是用逗号分割开的多个参数：

第一次访问找不到页面后，相当于会在服务器端产生一个 `doc_type=0&doc_id=22342566&doc_template=300458` 的查询，

而查询结果会生成的缓存的静态页面：0,22342566,300458.html

静态缓存的缺点：

- 复杂的触发更新机制：这两种机制在内容管理系统比较简单的时候都是非常适用的。但对于一个关系比较复杂的网站来说，页面之间的逻辑引用关系就成为一个非常复杂的问题。最典型的例子就是一条新闻要同时出现在新闻首页和相关的 3 个新闻专题中，在静态缓存模式中，每发一篇新文章，除了这篇新闻内容本身的页面外，还需要系统通过触发器生成多个新的相关静态页面，这些相关逻辑的触发也往往就会成为内容管理系统中最复杂的部分之一。
- 旧内容的批量更新：通过静态缓存发布的内容，对于以前生成的静态页面的内容很难修改，这样用户访问旧页面时，新的模板根本无法生效。

在动态缓存模式中，每个动态页面只需要关心，而相关的其他页面能自动更新，从而大大减少了设计相关页面更新触发器的需要。

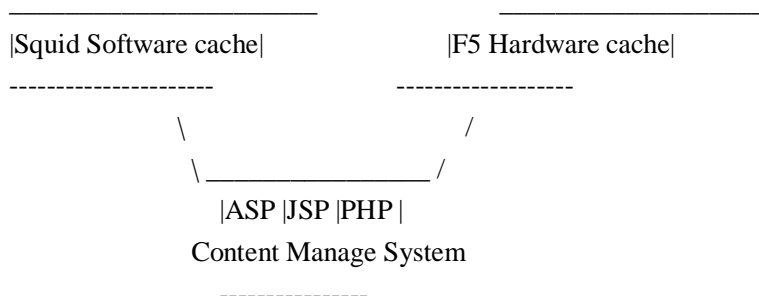
以前做小型应用的时候也用过类似方式：应用首次访问以后将数据库的查询结果在本地存成一个文件，下次请求时先检查本地缓存目录中是否有缓存文件，从而减少对后台数据库的访问。虽然这样做也能承载比较大的负载，但这样的内容管理和缓存管理一体的系统是很难分离的，而且数据完整性也不是很好保存，内容更新时，应用需要把相应内容的缓存文件删除。但是这样的设计在缓存文件很多的时候往往还需要将缓存目录做一定的分布，否则一个目录下的文件节点超过 3000，`rm *`都会出错。

这时候，系统需要再次分工，把复杂的内容管理系统分解成：内容输入和缓存这 2 个相对简单的系统实现。

- 后台：内容管理系统，专心的将内容发布做好，比如：复杂的工作流管理，复杂的

模板规则等.....

- 前台：页面的缓存管理则可以使用缓存系统实现



所以分工后：内容管理和缓存管理 2 者，无论哪一方面可选的余地都是非常大的：软件（比如前台 80 端口使用 SQUID 对后台 8080 的内容发布管理系统进行缓存），缓存硬件，甚至交给 [akamai](#) 这样的专业服务商。

面向缓存的站点规划

一个利用 SQUID 对多个站点进行做 WEB 加速 http acceleration 方案：

原先一个站点的规划可能是这样的：

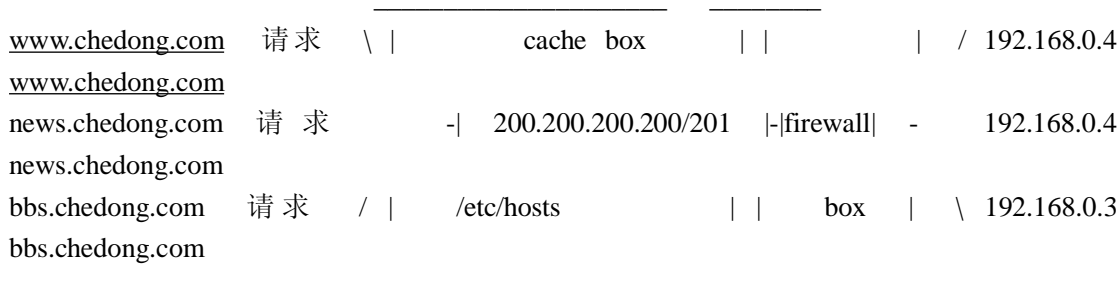
200.200.200.207 [www.chedong.com](#)

200.200.200.208 news.chedong.com

200.200.200.209 bbs.chedong.com

200.200.200.205 images.chedong.com

面向缓存服务器的设计中：所有站点都通过外部 DNS 指向到同一个 IP：200.200.200.200/201 这 2 台缓存服务器上（使用 2 台是为了冗余备份）



工作原理：

外部请求过来时，设置缓存根据配置文件进行转向解析。这样，服务器请求就可以转发到我们指定的内部地址上。

在处理多虚拟主机转向方面：mod_proxy 比 squid 要简单一些：可以把不同服务转向后台多个 IP 的不同端口上。

而 squid 只能通过禁用 DNS 解析，然后根据本地的/etc/hosts 文件根据请求的域名进行地址转发，后台多个服务器必须使用相同的端口。

使用反向代理加速，我们不仅可以得到性能上的提升，而且还能获得额外的安全性和配置的灵活性：

- 配置灵活性提高：可以自己在内部服务器上控制后台服务器的 DNS 解析，当需要在服务器之间做迁移调整时，就不用大量修改外部 DNS 配置了，只需要修改内部 DNS 实现服务的调整。
- 数据安全性增加：所有后台服务器可以很方便的被保护在防火墙内。
- 后台应用设计复杂程度降低：原先为了效率常常需要建立专门的图片服务器

images.chedong.com 和负载比较高的应用服务器 bbs.chedong.com 分离, 在反向代理加速模式中, 所有前台请求都通过缓存服务器: 实际上就都是静态页面, 这样, 应用设计时就不用考虑图片和应用本身分离了, 也大大降低了后台内容发布系统设计的复杂程度, 由于数据和应用都存放在一起, 也方便了文件系统的维护和管理。

基于 Apache mod_proxy 的反向代理缓存加速实现

Apache 包含了 mod_proxy 模块, 可以用来实现代理服务器, 针对后台服务器的反向加速
安装 apache 1.3.x 编译时:

```
--enable-shared=max --enable-module=most
```

注: Apache 2.x 中 mod_proxy 已经被分离成 mod_proxy 和 mod_cache: 同时 mod_cache 有基于文件和基于内存的不同实现

创建 /var/www/proxy, 设置 apache 服务所用户可写

mod_proxy 配置样例: 反相代理缓存+缓存

架设前台的 www.example.com 反向代理后台的 www.backend.com 的 8080 端口服务。

修改: httpd.conf

```
<VirtualHost *>
```

```
ServerName www.example.com
```

```
ServerAdmin admin@example.com
```

```
# reverse proxy setting
```

```
ProxyPass / http://www.backend.com:8080/
```

```
ProxyPassReverse / http://www.backend.com:8080/
```

```
# cache dir root
```

```
CacheRoot "/var/www/proxy"
```

```
# max cache storage
```

```
CacheSize 50000000
```

```
# hour: every 4 hour
```

```
CacheGcInterval 4
```

```
# max page expire time: hour
```

```
CacheMaxExpire 240
```

```
# Expire time = (now - last_modified) * CacheLastModifiedFactor
```

```
CacheLastModifiedFactor 0.1
```

```
# default expire tag: hour
```

```
CacheDefaultExpire 1
```

```
# force complete after percent of content retrived: 60-90%
```

```
CacheForceCompletion 80
```

```
CustomLog /usr/local/apache/logs/dev_access_log combined
```

```
</VirtualHost>
```

基于 Squid 的反向代理加速实现

Squid 是一个更专用的代理服务器, 性能和效率会比 Apache 的 mod_proxy 高很多。

如果需要 combined 格式日志补丁:

<http://www.squid-cache.org/mail-archive/squid-dev/200301/0164.html>

squid 的编译:

```
./configure
```

```
--enable-useragent-log
```

```
--enable-referer-log
```

```
--enable-default-err-language=Simplify_Chinese \ --enable-err-languages="Simplify_Chinese  
English" --disable-internal-dns
```



```
make
#make install
#cd /usr/local/squid
make dir cache
chown squid.squid *
vi /usr/local/squid/etc/squid.conf
在/etc/hosts 中： 加入内部的 DNS 解析， 比如：
192.168.0.4 www.chedong.com
192.168.0.4 news.chedong.com
192.168.0.3 bbs.chedong.com
-----cut here-----
# visible name
visible_hostname cache.example.com
# cache config: space use 1G and memory use 256M
cache_dir ufs /usr/local/squid/cache 1024 16 256
cache_mem 256 MB
cache_effective_user squid
cache_effective_group squid

http_port 80
httpd_accel_host virtual
httpd_accel_single_host off
httpd_accel_port 80
httpd_accel_uses_host_header on
httpd_accel_with_proxy on
# accelerator my domain only
acl acceleratedHostA dstdomain .example1.com
acl acceleratedHostB dstdomain .example2.com
acl acceleratedHostC dstdomain .example3.com
# accelerator http protocol on port 80
acl acceleratedProtocol protocol HTTP
acl acceleratedPort port 80
# access arc
acl all src 0.0.0.0/0.0.0.0
# Allow requests when they are to the accelerated machine AND to the
# right port with right protocol
http_access allow acceleratedProtocol acceleratedPort acceleratedHostA
http_access allow acceleratedProtocol acceleratedPort acceleratedHostB
http_access allow acceleratedProtocol acceleratedPort acceleratedHostC
# logging
emulate_httpd_log on
cache_store_log none
# manager
acl manager proto cache_object
```

```
http_access allow manager all
cachemgr_passwd pass all
```

-----cut here-----

创建缓存目录:

```
/usr/local/squid/sbin/squid -z
```

启动 squid

```
/usr/local/squid/sbin/squid
```

停止 squid:

```
/usr/local/squid/sbin/squid -k shutdown
```

启用新配置:

```
/usr/local/squid/sbin/squid -k reconfig
```

通过 crontab 每天 0 点截断/轮循日志:

```
0 0 * * * (/usr/local/squid/sbin/squid -k rotate)
```

可缓存的动态页面设计

什么样的页面能够比较好的被缓存服务器缓存呢? 如果返回内容的 HTTP HEADER 中有 "Last-Modified"和"Expires"相关声明, 比如:

Last-Modified: Wed, 14 May 2003 13:06:17 GMT

Expires: Fri, 16 Jun 2003 13:06:17 GMT

前端缓存服务器在期间会将生成的页面缓存在本地: 硬盘或者内存中, 直至上述页面过期。因此, 一个可缓存的页面:

- 页面必须包含 Last-Modified: 标记
一般纯静态页面本身都会有 Last-Modified 信息, 动态页面需要通过函数强制加上, 比如在 PHP 中:

```
// always modified now
```

```
header("Last-Modified: " . gmdate("D, d M Y H:i:s") . " GMT");
```

- 必须有 Expires 或 Cache-Control: max-age 标记设置页面的过期时间:
对于静态页面, 通过 apache 的 mod_expires 根据页面的 MIME 类型设置缓存周期:
比如图片缺省是 1 个月, HTML 页面缺省是 2 天等。

```
<IfModule mod_expires.c>
```

```
ExpiresActive on
```

```
ExpiresByType image/gif "access plus 1 month"
```

```
ExpiresByType text/css "now plus 2 day"
```

```
ExpiresDefault "now plus 1 day"
```

```
</IfModule>
```

对于动态页面, 则可以直接通过写入 HTTP 返回的头信息, 比如对于新闻首页 index.php 可以是 20 分钟, 而对于具体的一条新闻页面可能是 1 天后过期。比如: 在 php 中加入了 1 个月后过期:

```
// Expires one month later
```

```
header("Expires: " . gmdate("D, d M Y H:i:s", time() + 3600 * 24 * 30) . " GMT");
```

- 如果服务器端有基于 HTTP 的认证, 必须有 Cache-Control: public 标记, 允许前台 ASP 应用的缓存改造 首先在公用的包含文件中(比如 include.asp)加入以下公用函数:

```
<%
```

```

' Set Expires Header in minutes
Function SetExpiresHeader(ByVal minutes)
    ' set Page Last-Modified Header:
    ' Converts date (19991022 11:08:38) to http form (Fri, 22 Oct 1999 12:08:38 GMT)
    Response.AddHeader "Last-Modified", DateToHTTPDate(Now())

    ' The Page Expires in Minutes
    Response.Expires = minutes

    ' Set cache control to external applications
    Response.CacheControl = "public"
End Function

' Converts date (19991022 11:08:38) to http form (Fri, 22 Oct 1999 12:08:38 GMT)
Function DateToHTTPDate(ByVal OleDATE)
    Const GMTdiff = #08:00:00#
    OleDATE = OleDATE - GMTdiff
    DateToHTTPDate = engWeekDayName(OleDATE) & _
        ", " & Right("0" & Day(OleDATE),2) & " " & engMonthName(OleDATE) & _
        " " & Year(OleDATE) & " " & Right("0" & Hour(OleDATE),2) & _
        ":" & Right("0" & Minute(OleDATE),2) & ":" & Right("0" & Second(OleDATE),2) & " GMT"
End Function

Function engWeekDayName(dt)
    Dim Out
    Select Case WeekDay(dt,1)
        Case 1:Out="Sun"
        Case 2:Out="Mon"
        Case 3:Out="Tue"
        Case 4:Out="Wed"
        Case 5:Out="Thu"
        Case 6:Out="Fri"
        Case 7:Out="Sat"
    End Select
    engWeekDayName = Out
End Function

Function engMonthName(dt)
    Dim Out
    Select Case Month(dt)
        Case 1:Out="Jan"
        Case 2:Out="Feb"
        Case 3:Out="Mar"
        Case 4:Out="Apr"
        Case 5:Out="May"
        Case 6:Out="Jun"
        Case 7:Out="Jul"
    End Select
    engMonthName = Out
End Function

```

```

        Case 8:Out="Aug"
        Case 9:Out="Sep"
        Case 10:Out="Oct"
        Case 11:Out="Nov"
        Case 12:Out="Dec"
    End Select
    engMonthName = Out
End Function
%>
然后在具体的页面中，比如 index.asp 和 news.asp 的“最上面”加入以下代码：HTTP Header
<!--#include file="../include.asp"-->
<%
'页面将被设置 20 分钟后过期
SetExpiresHeader(20)
%>

```

应用的缓存兼容性设计

经过代理以后，由于在客户端和服务之间增加了中间层，因此服务器无法直接拿到客户端的 IP，服务器端应用也无法直接通过转发请求的地址返回给客户端。但是在转发请求的 HTTP 头信息中，增加了 HTTP_X_FORWARDED_??? 信息。用以跟踪原有的客户端 IP 地址和原来客户端请求的服务器地址：

下面是 2 个例子，用于说明缓存兼容性应用的设计原则：

' 对于一个需要服务器名的地址的 ASP 应用：不要直接引用 HTTP_HOST/SERVER_NAME，判断一下是否有 HTTP_X_FORWARDED_SERVER

```

function getHostName ()
    dim hostName as String = ""
    hostName = Request.ServerVariables("HTTP_HOST")
    if not isDBNull(Request.ServerVariables("HTTP_X_FORWARDED_HOST")) then
        if len(trim(Request.ServerVariables("HTTP_X_FORWARDED_HOST"))) > 0 then
            hostName = Request.ServerVariables("HTTP_X_FORWARDED_HOST")
        end if
    end if
    return hostName
end function

```

// 对于一个需要记录客户端 IP 的 PHP 应用：不要直接引用 REMOTE_ADDR，而是要使用 HTTP_X_FORWARDED_FOR，

```

function getUserIP () {
    $user_ip = $_SERVER["REMOTE_ADDR"];
    if ($_SERVER["HTTP_X_FORWARDED_FOR"]) {
        $user_ip = $_SERVER["HTTP_X_FORWARDED_FOR"];
    }
}

```

注意: HTTP_X_FORWARDED_FOR 如果经过了多个中间代理服务器, 有何能是逗号分割的多个地址,

比如: 200.28.7.155,200.10.225.77 unknown,219.101.137.3

因此在很多旧的数据库设计中(比如 BBS) 往往用来记录客户端地址的字段被设置成 20 个字节就显得过小了。

经常见到类似以下的错误信息:

Microsoft JET Database Engine 错误 '80040e57'

字段太小而不能接受所要添加的数据的数量。试着插入或粘贴较少的数据。

/inc/char.asp, 行 236

原因就是在设计客户端访问地址时, 相关用户 IP 字段大小最好要设计到 50 个字节以上, 当然经过 3 层以上代理的几率也非常小。

如何检查目前站点页面的可缓存性(Cacheability)呢? 可以参考以下 2 个站点上的工具:

<http://www.ircache.net/cgi-bin/cacheability.py>

附: SQUID 性能测试试验

phpMan.php 是一个基于 php 的 man page server, 每个 man page 需要调用后台的 man 命令和很多页面格式化工具, 系统负载比较高, 提供了 Cache Friendly 的 URL, 以下是针对同样的页面的性能测试资料:

测试环境: Redhat 8 on Cyrix 266 / 192M Mem

测试程序: 使用 apache 的 ab(apache benchmark):

测试条件: 请求 50 次, 并发 50 个连接

测试项目: 直接通过 apache 1.3 (80 端口) vs squid 2.5(8000 端口: 加速 80 端口)

测试 1: 无 CACHE 的 80 端口动态输出:

ab -n 100 -c 10 <http://www.chedong.com:81/phpMan.php/man/kill/1>

This is ApacheBench, Version 1.3d <\$Revision: 1.2 \$> apache-1.3

Copyright (c) 1996 Adam Twiss, Zeus Technology Ltd,

<http://www.zeustech.net/>

Copyright (c) 1998-2001 The Apache Group, <http://www.apache.org/>

Benchmarking localhost (be patient).....done

Server Software:

Apache/1.3.23

Server Hostname: localhost

Server

Port:

80

Document Path:

/phpMan.php/man/kill/1

Document Length: 4655 bytes

Concurrency Level: 5

Time taken for tests: 63.164 seconds

Complete requests: 50
Failed requests: 0
Broken pipe errors: 0
Total transferred: 245900 bytes
HTML transferred: 232750 bytes
Requests per second: 0.79 [#/sec] (mean)
Time per request: 6316.40 [ms]
(mean)
Time per request: 1263.28 [ms]
(mean, across all concurrent requests)
Transfer rate:
3.89 [Kbytes/sec] received

Connnection Times (ms)

	min	mean[+/-sd]	median	max
Connect:	0			
29	106.1	0	553	
Processing:	2942	6016		
1845.4	6227	10796		

Waiting:
2941 5999 1850.7 6226 10795

Total:
2942 6045 1825.9 6227 10796

Percentage of the requests served within a certain time (ms)

50%	6227
66%	7069
75%	7190
80%	7474
90%	8195
95%	8898
98%	9721
99%	10796
100%	10796 (last request)

测试 2: SQUID 缓存输出

/home/apache/bin/ab -n50 -c5

"<http://localhost:8000/phpMan.php/man/kill/1>"

This is ApacheBench, Version 1.3d <\$Revision: 1.2 \$> apache-1.3

Copyright (c) 1996 Adam Twiss, Zeus Technology Ltd,

<http://www.zeustech.net/>

Copyright (c) 1998-2001 The Apache Group, <http://www.apache.org/>

Benchmarking localhost (be patient).....done

Server Software:

Apache/1.3.23

Server Hostname: localhost

Server

Port:

8000

Document Path:

/phpMan.php/man/kill/1

Document Length: 4655 bytes

Concurrency Level: 5

Time taken for tests: 4.265 seconds

Complete requests: 50

Failed requests: 0

Broken pipe errors: 0

Total transferred: 248043 bytes

HTML transferred: 232750 bytes

Requests per second: 11.72 [#/sec] (mean)

Time per request: 426.50 [ms] (mean)

Time per request: 85.30 [ms] (mean,
across all concurrent requests)

Transfer rate:

58.16 [Kbytes/sec] received

Connnection Times (ms)

min	mean[+/-sd]	median	max
-----	-------------	--------	-----

Connect:

0	1		
9.5	0	68	

Processing:

7	83	537.4	
7	3808		

Waiting:

5	81	529.1	
6	3748		

Total:

7	84	547.0	
---	----	-------	--

Percentage of the requests served within a certain time (ms)

50%	7
66%	7
75%	7
80%	7
90%	7
95%	7
98%	8
99%	3876
100%	3876 (last request)

结论: $\text{No Cache} / \text{Cache} = 6045 / 84 = 70$

结论: 对于可能被缓存请求的页面, 服务器速度可以有 2 个数量级的提高, 因为 SQUID 是把缓存页面放在内存里的 (因此几乎没有硬盘 I/O 操作)。

小节:

- 大访问量的网站应尽可能将动态网页生成静态页面作为缓存发布, 甚至对于搜索引擎这样的动态应用来说, 缓存机制也是非常非常重要的。
- 在动态页面中利用 HTTP Header 定义缓存更新策略。
- 利用缓存服务器获得额外的配置和安全性
- 日志非常重要: SQUID 日志缺省不支持 COMBINED 日志, 但对于需要 REFERER 日志的这个补丁非常重要:
<http://www.squid-cache.org/mail-archive/squid-dev/200301/0164.html>

参考资料:

[HTTP 代理缓存](#)

<http://vancouver-webpages.com/proxy.html>

可缓存的页面设计

<http://linux.oreillynet.com/pub/a/linux/2002/02/28/cachefriendly.html>

运用 ASP.NET 的输出缓冲来存储动态页面 - 开发者 - ZDNet China

<http://www.zdnet.com.cn/developer/tech/story/0,2000081602,39110239-2,00.htm>

相关 RFC 文档:

- [RFC 2616](#):
 - [section 13 \(Caching\)](#)
 - [section](#)

- 14.9 (Cache-Control header)
 - o [section](#)
- 14.21 (Expires header)
 - o [section](#)
- 14.32 (Pragma: no-cache) is important if you are interacting with HTTP/1.0 caches
 - o [section](#)
- 14.29 (Last-Modified) is the most common validation method
 - o [section](#)
- 3.11 (Entity Tags) covers the extra validation method

可缓存性检查

<http://www.web-caching.com/cacheability.html>

缓存设计要素

<http://vancouver-webpages.com/CacheNow/detail.html>

ZOPE 上的几篇使用 APACHE MOD_PROXY MOD_GZIP 加速的文档

http://www.zope.org/Members/anser/apache_zserver/

http://www.zope.org/Members/softsign/ZServer_and_Apache_mod_gzip

<http://www.zope.org/Members/rbeer/caching>

I 开发大型高负载类网站应用的几个要点[nightsailer]

大 | 中 | 小

2007/05/17
 14:12
 772
 huzhangyou2002
 信仰的服务器设计

作者: nightsailer

来源: <http://www.phpchina.com/bbs/thread-15484-1-1.html>

看了一些人的所谓大型项目的方法,我感觉都是没有说到点子上,有点难受。我也说说自己的看法.我个人认为,很难衡量所谓项目是否大型,即便很简单的应用在高负载和高增长情况下都是一个挑战.因此,按照我的想法,姑且说是高负载高并发或者高增长情况下,需要考虑的问题.这些问题,很多是和程序开发无关,而是和整个系统的架构密切相关的。

数据库

没错,首先是数据库,这是大多数应用所面临的首个 SPOF。尤其是 Web2.0 的应用，数据库的响应是首先要解决的。

一般来说 MySQL 是最常用的，可能最初是一个 mysql 主机，当数据增加到 100 万以上，那么，MySQL 的效能急剧下降。常用的优化措施是 M-S（主-从）方式进行同步复制，将查询和操作和分别在不同的

服务器上进行操作。我推荐的是 M-M-Slaves 方式，2 个主 Mysql，多个 Slaves，需要注意的是，虽然有 2 个 Master，

但是同时只有 1 个是 Active，我们可以在一定时候切换。之所以用 2 个 M，是保证 M 不会又成为系统的 SPOF。Slaves 可以进一步负载均衡，可以结合 LVS，从而将 select 操作适当的平衡到不同的 slaves 上。

以上架构可以抗衡到一定量的负载，但是随着用户进一步增加，你的用户表数据超过 1 千万，这时那个 M 变成了

SPOF。你不能任意扩充 Slaves，否则复制同步的开销将直线上升，怎么办？我的方法是表分区，从业务层面上进行分区。最简单的，以用户数据为例。根据一定的切分方式，比如 id，切分到不同的数据库集群去。

全局数据库用于 meta 数据的查询。缺点是每次查询，会增加一次，比如你要查一个用户 nightsailer，你首先要到全局数据库群找到 nightsailer 对应的 cluster id，然后再到指定的 cluster 找到 nightsailer 的实际数据。

每个 cluster 可以用 m-m 方式，或者 m-m-slaves 方式。

这是一个可以扩展的结构，随着负载的增加，你可以简单的增加新的 mysql cluster 进去。

需要注意的是：

- 1、禁用全部 auto_increment 的字段
- 2、id 需要采用通用的算法集中分配
- 3、要具有比较好的方法来监控 mysql 主机的负载和服务的运行状态。如果你有 30 台以上的 mysql 数据库在跑就明白我的意思了。
- 4、不要使用持久性链接（不要用 pconnect），相反，使用 sqlrelay 这种第三方的数据库链接池，或者干脆自己做，因为 php4 中 mysql 的链接池经常出问题。

缓存

缓存是另一个大问题，我一般用 memcached 来做缓存集群，一般来说部署 10 台左右就差不多（10g 内存池）。需要注意一点，千万不能用使用 swap，最好关闭 linux 的 swap。

负载均衡/加速

可能上面说缓存的时候，有人第一想的是页面静态化，所谓的静态 html，我认为这是常识，不属于要点了。页面的静态化随之带来的是静态服务的

负载均衡和加速。我认为 Lighttpd+Squid 是最好的方式了。

LVS <----->lighttpd====>squid(s) ===>lighttpd

上面是我经常用的。注意，我没有用 apache，除非特定的需求，否则我不部署 apache，因为我一般用 php-fastcgi 配合 lighttpd，性能比 apache+mod_php 要强很多。

squid 的使用可以解决文件的同步等等问题，但是需要注意，你要很好的监控缓存的命中率，尽可能的提高的 90%以上。

squid 和 lighttpd 也有很多的话题要讨论，这里不赘述。

存储

存储也是一个大问题，一种是小文件的存储，比如图片这类。另一种是大文件的存储，比如搜索引擎的索引，一般单文件都超过 2g 以上。小文件的存储最简单的方法是结合 **lighttpd** 来进行分布。或者干脆使用 **Redhat** 的 **GFS**，优点是应用透明，缺点是费用较高。我是指

你购买盘阵的问题。我的项目中，存储量是 2-10Tb，我采用了分布式存储。这里要解决文件的复制和冗余。这样每个文件有不同的冗余，这方面可以参考 **google** 的 **gfs** 的论文。

大文件的存储，可以参考 **nutch** 的方案，现在已经独立为 **hadoop** 子项目。(你可以 **google it**)

其他:

此外, **passport** 等也是考虑的, 不过都属于比较简单的了。

吃饭了, 不写了, 抛砖引玉而已。

【回复】

9tmd :

说了关键的几个部分, 还有一些比如 **squid** 群、**LVS** 或者 **VIP** (四层交换) 之类的必须考虑, 数据库逻辑分表不需要 **master** 里面查 **id**, 可以定期缓存或者程序逻辑上进行控制。

跟大家分享一下我的经验: <http://www.toplee.com/blog/archives/337.html> (欢迎讨论)

nightsailer:

楼上说的很好.

我再说一下关于为何要在主表查询, 最主要的因素是考虑到复制和维护的问题。假设按照程序逻辑, 用户 **nightsailer** 应该在 **s1** 集群, 但是由于种种原因, 我须要将 **nightsailer** 的数据从 **s1** 集群转移到 **s5** 集群或者某些时候, 我需要将某几个集群的数据合并, 此时, 我维护的时候只需要更新一下主数据库中 **nightsailer** 的 **cluster id** 从 1 变成 5, 维护的工作可以独立进行, 无需考虑更新应用程序的逻辑。也许程序的 **id** 分配逻辑可以考虑到这种情况, 但是这样一来, 你的这个逻辑会发散到各个应用中, 产生的代码的耦合是很高的。相反, 采用查表这种方式, 只需要在最初的时候进行初始分配, 那么其他的应用是无需考虑这些算法和逻辑的。

当然, 我最初提到的增加这次查询并不是说每次查询都需要找主数据库, 缓存策略是必定要考虑的。

至于说为什么要禁用 **auto_increment**, 我想也清楚了, 数据的合并和分隔, 肯定是不能用 **auto_increment** 的。

nightsailer:

在闲扯一下, **PHP** 的优化可以有很多, 主要的措施:

1、使用 **FCGI** 方式, 配合 **lighttpd**, **Zeus**.

我个人比较喜欢 **Zeus**, 简单可靠。不过, 需要 ¥ ¥ ¥。

lighty 也不错, 配置文件也很简单, 比较清爽。最新的 1.5, 虽然不稳定, 但是配合 **linux** 的 **aio**, 性能的提升非常明显。即便现在的稳定版, 使用 2.6 的 **epoll** 可以得到的性能是非常高。当然, **lighty** 比 **zeus** 缺点是对 **smp**

的支持很有限, 所以可以采用多服务器负载, 或者干脆起不同的进程服务监听不同的端口。

2、专门的 **PHP FCGI** 服务器。

好处多多, 在这个服务器上, 就跑 **php** 的 **fcgi** 服务, 你可以把一些缓存加上, 比如 **xcache**, 我个人喜欢这个。

还有别的，套用大腕的话，把能装的都装上，呵呵。

另外，最主要的是，你可以只维护一个 **php** 的环境，这个环境能够被 **apache,zeus,lighttpd** 同时 share，前提是这些都使用 **php** 的 **fcgi** 模式，而且，**xcache** 可以充分发挥！

3、apache+mod_fastcgi

apache 并非无用，有时候也需要。比如我用 **php** 做了一个 **web_dav** 的服务器，在其他有问题，只能跑 **apache**。那么，**apache** 安装一下 **mod_fastcgi**，通过使用 **external server**，使用 2 配置的 **php fastcgi**。

4、优化编译

ICC 是我的首选，就是 **intel** 的编译器啦，用 **icc** 重新编译 **php,mysql,lighty**，能编的都编，会有不小的收获的。尤其是你用

intel 的 **cpu** 的话。

5、php4 的 64 位需要 patch

好像没有人在 **linux x86_64** 上编译过 **php4** 吧，我曾经 **googleit**

,别说国内了，连老外都很少用。

这里就做个提醒把，如果用 **php** 官方下载的(包括最新的 **php-4.4.4**)，统统无法编译通过。问题是出在 **autoconf** 上，需要

手工修改 **config.m4**，一般是在 **mysql,gd,ldap** 等一些关键的 **extension** 上，还有 **phpize** 的脚本。把 **/usr/lib64** 加入到

config.m4 中相关搜索的 **path** 中。

不过我估计很少人像我这样死用 **php4** 不防，呵呵。**php5** 就没有问题。

我也考虑正在迁移到 **php5.2**，写代码太方便了，一直忍着呢。

nightsailer:

QUOTE:

原帖由 **wuexp** 于 2007-1-3 17:01 发表

分表会使操作数据(更改,删除,查询)变的很复杂,特别是遇到排序的时候就更麻烦了.

曾经考虑根据用户 **id** 哈希一下,插入到相应的分表里

明白你的意思。

不过我们可能讨论的不完全一样，呵呵。

我所说的分表要依据不同的业务情况来划分的，

1、可以是垂直划分，

比如依据业务实体切分，比如用户 **a** 的 **blog** 帖子，用户的 **tag**，用户的评论都在 **a** 数据库 **u**，甚至是完整的一套数据结构(这种情况下应该说是分数据库)

2、也可以水平划分，

一个表的数据分在不同的数据库上。

比如 **message** 表，你可能分为 **daily_message,history_message**，

dialy_meesage 可能是 **hot** 对象，**week_message** 是 **warm**，2 个月以前的帖子

可能属于 **cold** 对象了。这些对象依据访问频度不同会划分到不同的数据库群上。

3、二者结合

不过，不论如何，更改、删除并不复杂，和未分区的表没有区别。

至于查询和排序，不可能仅仅是通过 `select, order` 吧？

而是应该产生类似摘要表，索引表，参考表。。。

另外，要根据业务具体分析减少垃圾数据，有些时候，只需要最初的 1 万条记录，那么所有表数据的排序就不需要了。很多传统的业务，比如零售，流水表很大，但是报表的数据并非实时生成的，扎报表应该不陌生。

也可以参考很多网站的做法，比如 `technorati` 啊, `flickr` 之类的。

所谓的麻烦是你设计系统的结构的时候要考虑到，在设计数据库的时候更要注意，因此只要项目的 `framework` 最初设计比较完备，那么可以说大部分对开发人员是透明的。前提是，你一定要设计好，而不是让程序员边写代码边设计，那会是噩梦。

我写这么多废话，并非仅仅是对程序员来说，也许对设计者更有用。

9tmd :

程序逻辑上控制表拆分只需要维护一个数据库访问的配置文件即可，对于开发来说，完全透明，可以不用关心访问的是哪里，而只需要调用通用的接口即可，曾经做过的系统里面，这样的应用经常遇到，尤其在 `passport`、社区帖子等方面的处理上应用最多。

原来在 `yahoo` 工作和后来 `mop` 工作都使用了这样的架构，整体感觉来说还是值得信赖的，单表毕竟存在面对极限数据量的风险。

9tmd :

前面老是有人问 `auto_increment` 的问题，其实这是 `MySQL` 官方专门针对 `M/S` 的 `Replication` 做过的说明，因为 `MySQL` 的同步是依靠同步 `MySQL` 的 `SQL` 日志来实现的，事实上单向的 `Master->Slave` 使用 `auto_increment` 是没有问题的，而双向的 `M/M` 模式就会存在问题了，稍微一思考就知道怎么回事了。官方文档：

[http://dev.mysql.com/tech-resour ... ql-replication.html](http://dev.mysql.com/tech-resour...ql-replication.html)

[http://dev.mysql.com/doc/refman/ ... auto-increment.html](http://dev.mysql.com/doc/refman/...auto-increment.html)

另外，在使用 `MySQL` 的同步时，需要注意在自己的代码里面，写 `SQL` 的时候不要使用 `MySQL` 自己提供的类似 `NOW()` 之类的函数，而应该使用 `php` 程序里面计算的时间带入 `SQL` 语句里面，否则同步的时候也可能导致值不相等，这个道理可以牵涉出另外一些类似的问题，大家可以考虑一下。

参考文章：

<http://blog.csdn.net/heiyeshuwu/archive/2007/01/04/1473941.aspx>

<http://www.phpchina.com/bbs/thread-15484-1-1.html>

I Memcached 和 Lucene 笔记

By Michael

前段时间完成的项目使用了大量的 Memcached，整个架构在性能上的确提高了很多，的确不是一点点的提高，面向大负载访问的时候，MySQL 数据库仍然可以做到轻量级的负载，效果不错，建议有条件的朋友一定要把项目改造到 Memcached 上，著名的 Vbb 论坛当前的版本就已经开始支持使用 Memcached 进行论坛数据缓存。我原来在 MOP 的时候，我们也大量的采用这个东西。

在使用 Memcached 方面，谈不上什么经验，反正极端的性能最大化就是使用永久的缓存，通过你的程序逻辑去控制和维护 MC 里面的缓存数据，我做的项目就是这样处理的，程序的逻辑的确增加了复杂度，但是对于商业项目来说，这种付出是非常值得的。

Memcached 唯一可能需要注意的是，他对 key 的操作不是原子级别的，所以在高并发处理的时候，对同一个 key 的写操作可能会导致覆盖，这个需要自己从程序逻辑上进行处理，这个理论我并没有深入研究，不过 JH 看了源代码给了我这样的结论，按照 JH 的实力和人品，我认为有 80% 以上的可信度：)

对于 Lucene，大部分人不陌生，相关的技术也不用太多讲解，网上到处都是相关的文档。我最近想通过 PHP 来找到一个最佳的整合 Lucene 的方法，并且应用到正规的商业应用中，目前知道的可选方案是 Pecl 的 Clucene 模块和 Zend Framework 的 Zend_Search_Lucene 模块，这两个东西目前我使用的感觉都不算太好，另外还有一种是使用 PHP 的 Java 扩展支持（有两种，一种是 php_java 扩展，一种是 php_java 的 bradage 方式），这个感觉也比较怪异，最后还有一种知道的办法就是使用系统调用 java 命令执行 Lucene 功能。这个没有试过，不知性能可以达到什么程度。

在这里做个记号，等有了进一步的收获补进来。

I 使用开源软件，设计高性能可扩展网站

2006-6-17 于敦德

上次我们以 LiveJournal 为例详细分析了一个小网站在一步一步的发展成为大规模的网站中性能优化的方案，以解决在发展中由于负载增长而引起的性能问题，同时在设计网站架构的时候就从根本上避免或者解决这些问题。

今天我们来看一下在网站的设计上一些通常使用的解决大规模访问，高负载的方法。我们将主要涉及到以下几方面：

- 1、 前端负载
- 2、 业务逻辑层

3、数据层

在 [LJ 性能优化文章](#) 中我们提到对服务器分组是解决负载问题，实现无限扩展的解决方案。通常中我们会采用类似 LDAP 的方案来解决，这在邮件的服务器以及个人网站，博客的应用中都有使用，在 Windows 下面有类似的 Active Directory 解决方案。有的应用（例如博客或者个人网页）会要求在二级域名解析的时候就将用户定位到所属的服务器群组，这个时候请求还没到应用上面，我们需要在 DNS 里解决这个问题。这个时候可以用到一款软件 [bind dlz](#)，这是 bind 的一个插件，用于取代 bind 的文本解析配置文件。它支持包括 LDAP，BDB 在内的多种数据存储方式，可以比较好的解决这个问题。

另外一种涉及到 DNS 的问题就是目前普遍存在的南北互联互通的问题，通过 bind9 内置的视图功能可以根据不同的 IP 来源解析出不同的结果，从而将南方的用户解析到南方的服务器，北方的用户解析到北方的服务器。这个过程中会碰到两个问题，一是取得南北 IP 的分布列表，二是保证南北服务器之间的通讯顺畅。第一个问题有个笨办法解决，从日志里取出所有的访问者 IP，写一个脚本，从南北的服务器分别 ping 回去，然后分析结果，可以得到一个大致准确的列表，当然最好的办法还是直到从运营商那里拿到这份列表。后一个问题解决办法比较多，最好的办法就是租用双线机房，同一台机器，双 IP，南北同时接入，差一些的办法就是南北各自找机房，通过大量的测试找出中间通讯顺畅的两个机房，后一种通常来说成本较低，但效果较差，维护不便。

另外 DNS 负载均衡也是广泛使用的一种负载均衡方法，通过并列的多条 A 记录将访问随即的分布到多台前端服务器上，这种通常使用在静态页面居多的应用上，几大门户内容部分的前端很多都是用的这种方法。

用户被定位到正确的服务器群组后，应用程序就接手用户的请求，并开始沿着定义好的业务逻辑进行处理。这些请求主要包括两类静态文件(图片，js 脚本,css 等)，动态请求。

静态请求一般使用 squid 进行缓存处理，可以根据应用的规模采用不同的缓存配置方案，可以是一级缓存，也可以是多级缓存，一般情况下 cache 的命中率可以达到 70%左右，能够比较有效的提升服务器处理能力。Apache 的 deflate 模块可以压缩传输数据，提高速度，2.0 版本以后的 cache 模块也内置实现磁盘和内存的缓存，而不必要一定做反向代理。

动态请求目前一般有两种处理方式，一种是静态化，在页面发生变化时重新静态页面，现在大量的 CMS，BBS 都采用这种方案，加上 cache，可以提供较快的访问速度。这种通常是写操作较少的应用比较适合的解决方案。

另一种解决办法是动态缓存，所有的访问都仍然通过应用处理，只是应用处理的时候会更多的使用内存，而不是数据库。通常访问数据库的操作是极慢的，而访问内存的操作很快，至少是一个数量级的差距，使用 [memcached](#) 可以实现这一解决方案，做的好的 memcache 甚至可以达到 90% 以上的缓存命中率。10 年前我用的还是 2M 的内存，那时的一本杂事上曾经风趣的描述一对父子的对话：

儿子：爸爸，我想要 1G 的内存。

爸爸：儿子，不行，即使是你过生日也不行。

时至今日，大内存的成本已经完全可以承受。Google 使用了大量的 PC 机建立集群用于数据处理，而我一直觉得，使用大内存 PC 可以很低成本的解决前端甚至中间的负载问题。由于 PC 硬盘寿命比较短，速度比较慢，CPU 也稍慢，用于做 web 前端既便宜，又能充分发挥大内存的优势，而且坏了的话只需要替换即可，不存在数据的迁移问题。

下面就是应用的设计。应用在设计的时候应当尽量设计成支持可扩展的数据库设计，数据库可以动态的添加，同时支持内存缓存，这样的成本是最低的。另外一种应用设计的方法是采用中间件，例如 [ICE](#)。这种方案的优点是前端应用可以设计的相对简单，数据层对于前端应用透明，由 ICE 提供，数据库分布式的设计在后端实现，使用 ICE 封装后给前端应用使用，这路设计对每一部分设计的要求较低，将业务更好的分层，但由于引入了中间件，分了更多层，实现起来成本也相对较高。

在数据库的设计上一方面可以使用集群，一方面进行分组。同时在细节上将数据库优化的原则尽量应用，数据库结构和数据层应用在设计上尽量避免临时表的创建、死锁的产生。数据库优化的原则在网上比较常见，多 google 一下就能解决问题。在数据库的选择上可以根据自己的习惯选择，Oracle，MySQL 等，并非 Oracle 就够解决所有的问题，也并非 MySQL 就代表小应用，合适的就是最好的。

前面讲的都是基于软件的性能设计方案，实际上硬件的良好搭配使用也可以有效的降低时间成本，以及开发维护成本，只是在这里我们不再展开。

网站架构的设计是一个整体的工程，在设计的时候需要考虑到性能，可拓展性，硬件成本，时间成本等等，如何根据业务的定位，资金，时间，人员的条件设计合适的方案是件比较困难的事情，但多想多实践，终究会建立一套适合自己的网站设计理念，用于指导网站的设计工作，为网站的发展奠定良好的基础。

I 面向高负载的架构

Lighttpd+PHP(FastCGI)+Memcached+Squid

By Michael

因新项目，开始从 Apache 上转移到 Lighttpd 上，同时还有 Memcached 的大量使用，借此机会把 toplee.com 的服务器环境也进行一些改造，顺便整理一份文档留存！

更多大型架构的经验，可以看我之前的一篇 blog：
<http://www.toplee.com/blog/archives/71.html>

12.31 截至今天完成以下内容：

1. 完成 lighttpd 的安装配置，并且做了大量的优化；
2. 几乎全部看完了 <http://trac.lighttpd.net/trac/wiki> 上的文档；
3. 配置了 lighttpd 和 php 的 fastcgi 支持；
4. 增加了 php 对 XCache 的支持；
5. 设置了部分域名在 lighttpd 上的解析；
6. 完成了 Apache 通过 mod_rewrite 和 mod_proxy 将部分域名以及全部的 php 访问转到 lighttpd 上；
7. 完成 Memcached 的环境搭建，并且修改了部分数据库操作缓存到 MC 上；

效果：

1. 系统负载变低了不少，响应速度得到提升；
2. MC 的效果非常理想，数据库压力得到很大减轻。

TODO:

（下面的事情等我买了第二台服务器后进行，目前仅在帮朋友的项目上这么干了）

- 配置 MySQL 的 Master/Slave 模式，把对数据库的 Write 和 Read 进行分开
- 加入 squid 群进行缓存加速
- 其他（比如 DNS 负载均衡加 LVS 的四层交换...）

To be continued...

注册 | 登录 | 发表文章

I 思考高并发高负载网站的系统架构

2007-04-13 10:38:10

大中小

下面是我 10 月中旬的想法，经过和小黑的讨论，现在想法有些变化。

如今百家*店的网站架构已经在超负荷工作了。服务器经常达到 100% 的使用率。主要是数据库占用了大量的 CPU 资源。这样的系统，根本无法跟上网站的发展。

所以，我针对我们网站，考虑了一些关于网站流量分流的方法。

1，看了一下别人的文章，大部分都说，现在的网站瓶颈在于数据库。所以，我们这里放在第一条。我们设计的网站要求用户数量要达到 1 亿。（目前淘宝用户近 2000 万，腾讯用户近 10 亿，活跃用户 1 亿多）。当然，我们的设计要多考虑一些，所以，就定位在淘宝的用户数和腾讯的用户数之间。

用户名列表单独建立数据库，以便随时将此数据库独立出去单独建服务器。用户登录的时候，数据库要从 1 亿记录中查找数据，即使使用主索引，也要将近 1 秒时间（在 SQLSERVER 中，使用 like 查找 20 万条数据，就需要大约 2 秒钟，这里是按照精确匹配以及主索引联合的方式查找）。所以，我们要将用户表分表。以 26 个字母为分表顺序，中文开头的用户名一张表，数字开头的用户名一张表。这样就有 28 张表，平均下来，一张表 300 万数据，最多的一张表估计大约 1000 万数据。然后，以用户名为主索引，SQLSERVER 数据库应该可以应付过来了。

除了用户表单独建立数据库以外，还要准备大城市大约需要 100 万左右的商品数据，500 万左右的帖子（目前杭州网论坛有超过 30 万主题贴，600 万回复贴），所以，现将商品数据存放在一个独立的数据库中，论坛帖子也存放在一个独立的数据库中，商品数据按城市分表，论坛帖子也按城市分，分别是主题贴一张表，回复贴一张表。

商店数量（淘宝目前有约 60 万活跃商店）我们网站应为不释放商店，所以需要更多的表存储该数据，目前无法确定，约为 1000 万家商店。保险期间，也独立建一个数据库，到时候可以和其他小数据库同用一个服务器。

商店对应的商店分类，以及友情链接，由于数量要预定至少 10 倍于商店数量，所以也要分别单独建立数据库，并按照用户名分表。

网站总设置，以及城市列表，版主信息等，这些可以生成静态内容的单独建立一个数据库。城市内商品分类需要单独建立一个数据库。并按照城市分表。

其他内容同理。到这里为止，理论上解决了数据库的瓶颈问题。

2，尽可能生成静态 HTML 页面。首页生成 HTML 页面，城市首页生成 HTML 页面，所有商品页面生成 HTML 页面，帖子第一页（前 10 篇）也生成 HTML 页面。如果一个主题贴超过 1 页，可以点击“更多”查看。这样可以节省服务器资源。生成页面的时候，服务器会占用大量 CPU 资源。所以，此功能要单独放置在一个独立的服务器中，并在该服务器上建立一个缓存队列数据库，用户在提交表单的时候，将数据保存在缓存队列数据库中，等待服务器的处理。服务器按照发表时间顺序（主索引）处理这些内容。将生成的 HTML 商品页面放在一个文件夹内（可随时增设服务器），上传图片和处理图片，存放图片在另一个文件夹内（图片最消耗 IIS 资源，以后一定要增设图片服务器）。并将处理好的内容存入主数据库中，并在缓存队列数据库中删除处理好的记录。这 2 块是占用 CPU 的大头，要随时准备移除主 WEB 服务器。

3，使用缓存。有些网页，像首页，可以学习百度首页，缓存 24 小时。

4，读取和写入数据库分开。用 2 个完全一样的数据库，一个专门用来写入，一个用来

读取，隔断时间将新加入的数据从写入数据库拷贝到读取数据库，这样可以减少数据库的符合。这个方法听说很多网站都采用的。

- 5，在北方，教育网等特殊网络做镜像服务器；
- 6，使用负载均衡技术。由特殊的服务商提供方案。

I "我在 SOHU 这几年做的一些门户级别的程序系统(C/C++ 开发)"

Bserv:

用于高负载，高读写速度的单点和集合数据。内核为 BerkeleyDB，外壳为 UDP 线程池。接口为读写单点数据或者集合数据。单点数据就是 Key->Value 数据。集合数据就是有索引的数据，List->Keys->Values。比如一个班级所有成员，一个主贴所有回帖等等。DBDS 性能很高，每秒读取>800 个每秒，写>300 个每秒（志强 xeon:2G*2，72Gscsi，Ram:2G）配合 java 接口，目前应用在 ChinaRen 所有项目中（ChinaRen 校内，校友录，社区等等）。是整个 ChinaRen 的核心数据服务，大概配备了 50 台服务器。特点：高速，高请求量。用于各种数据的低成本存储，解决数据库无法实现超高速读写的问题。门户级别的高速数据服务。

OnlineServer:

ChinaRen/SOHU 小纸条系统核心 核心为 3 个小 server 系统：online2(在线系统业务逻辑)，userv(用户资料系统)，cserv(LRU 缓存) 这三个子系统都是 UDP+线程池结构，单进程+多线程。配备 java 接口，apache_mod 的 json 和 xml 接口。online2 包括了大部分业务逻辑，包括，上线，好友系统，纸条系统。userv 包括设置用户各种属性，信息。cserv 是个大的 lru 缓存，用于减小磁盘 IO。可以放各种信息块，包括用户信息，好友，留言等。目前配备 4 台服务器（DL380，xeon:3G*2，SCSI:146G raid，Ram:2G），用户分布到 4 台服务器上，相互交互。服务器可以由 1 台到 2 台，到 4 台，到 8 台。底层存储为文件存储（无数据库），用 reiserfs。配套系统：mod_online，两个版本，apache 和 lighttpd 版本，用于页面上显示蜡烛人。请求量巨大，目前用 lighttpd 版本的 mod_online。放在 sohu 的 squid 前端机器上，运行在 8080，大概 8 台，每台请求量大概 500-800 个每秒。蜡烛人在所有 ChinaRen 页面有 ID 的地方显示用户是否在线。目前这套在线系统，作为 SOHUIM 的内核原型。准备开发 WEBIM 系统，用户所有 SOHU 矩阵用户的联络。

apache_mod 系列:

基于 apache2 的服务有很多，用于高请求量，快速显示的地方。1.mod_gen_verifyimg2: 用于显示验证码，使用 GD2，freetype。直接在 apache 端返回 gif 流，显示随机的字体，角度，颜色等等。用于 ChinaRen 各个需要验证码的页面，请求量很大。2.mod_ip2loc 用于 apache 端的 IP->物理地址转换，高速，高效。读取数据文件到内部数据树，高速检索，获得客户端 ip 的物理地址。用于需要 IP 自动定位的产品，还有就是数据统计等。比如 ChinaRen 校内，每个客户端请求都能获得物理地址，用于应用的逻辑处理。

3.mod_pvserver2

ChinaRen 社区帖子点击的记录和显示。根据 URL，得到帖子 ID，通过 UDP 数据包，统计到 bserv 系统。并且把结果通过 Cookie 返回到客户端。html 直接用 javascript 显示点击数在帖子上。解决了点击数量高效记录，高效读取和非动态页面程序显示的问题。4.mod_online 用于 ChinaRen 页面上的蜡烛人显示。和 onlineserver 通讯，得到用户在线状态和其他状态信息。请求量很大，每台前端大概 500-800 个请求。5.其他 mod 还有一些认证的，访问统计的，特种 url 过虑跳转的，页面 key 生成的，还有若干。特点：高速，密集超高请求量。前端分担应用服务器压力，高效。

cserv:

高速 LRU 缓存系统。内核是 UDP+线程池+LRU 结构 (hash+PQueue)。用于存放各种数据块, Key->Value 结构。通过 LRU 方式提供给应用, 可减小文件 IO, 磁盘 IO 等慢速操作。目前用于 ChinaRen 在线系统的用户资料缓存。特点: 高速读写, 低成本。

ddap:

UDP+线程池, 单进程, 多线程的服务端程序原型, 大部分程序由这个结构开始。性能为 8000-10000 个请求每秒。

eserv:

访问统计系统 用于用户访问的次数和最后上线时间的存储和读写。用于 ChinaRen 校友录每个班级的访问记录。存储为文件存储, 并有同时写入后备的 bserv, 用于备份和检索。目前性能, 每台机器每秒 50 个记录, 100 个读每秒。能满足校友录巨大的用户登录记录的需要。特点: 无数据库, 纯文件存储, 高速读写。低成本

logserver:

用于各种事件的日志记录 核心为 ddap, UDP+线程池 功能是分模块记录各种日志。ChinaRen 所有用户服务, 系统日志, 都记录在 logserver 中。用于统计, 查询。写入性能很好, 每秒 100 个单台机器。特点: 高速高效, 低成本, 海量。

SessionServ:

session 系统 核心为 ddap, UDP+线程池 用于在内存中存储临时数据。有 get/put/del/inc 等操作。广泛的用于固定时间窗口的小数据存储。比如过期, 数据有效性检测, 应用 同步等等。由于是全内存操作, 所以速度很快, 存取速度应该>1000 个每秒。目前广泛用与 ChinaRen 社区, 校内, 校友录等业务当中。特点: 高速高效, 低成本, 应用广泛。

其他 server:

MO_dispatcher: 用于短信上行接口的数据转发, 使用 TCP。能高速大流量根据业务号码分发到各个应用服务中。目前用于 SOHU 短信到 ChinaRen 各短信服务的转发。sync: 用于静态前端同步, 分客户端和服务端程序。客户端通过 TCP 链接和服务端获取需要同步的文件列表, 并且通过 TCP 高速更新本地文件。此同步程序用于多客户端, 单服务端。比如一台服务器生成静态文件, 同步这些文件到若干客户前端去。特点: 门户级静态内容服务器间同步, 高效, 高速, 大流量。目前用于 ChinaRen 社区的静态帖子。

总结一下:

门户的核心服务, 要求是高效率, 高密度存取, 海量数据, 最好还是低成本。不要用数据库, 不要用 java, 不要用 mswin。用 C, 用内存, 用文件, 用 linux 就对了。

I 原中国顶级门户网站架构分析 1

首先声明, 下面的内容都是我个人根据一些工具形成的猜想。并不保证和现实中各大门户网站所用的架构一摸一样, 不过我认为八九不离十了^_^。

整篇文章我想分 2 个部分来讲: 第一部分是分析国内 2 大顶级门户网站首页和频道的初步的基本构架。第二部分我将自己做的实验文档记录下来。希望每个 SA 心里都能有这样的架构。

新浪和搜狐在国内的知名度可谓无人不知无人不晓。他们每天的点击率都在千万以上。这样大的访问量对于新浪和搜狐来说怎样利用有限的资源让网民获得最快的速度成为首要的前提, 毕竟现在网络公司已经离开了烧钱的阶段, 开始了良性发展, 每一笔钱砸下去都需要一定回响才行的。另一方面, 技术人员要绞尽脑汁, 不能让用户老是无法访问、或者访问速度极慢。这样就算有再

好的编辑、再好的销售，他们也很难将广告位卖出去，等待他们的将是关门。当然这些情况都没有发生，因为他们的技术人员都充分的利用了现有资源并将他们发挥到了极至。说到底就是用 **squid** 做 **web cache server**，而 **apache** 在 **squid** 的后面提供真正的 **web** 服务。当然使用这样的架构必须要保证主页上大部分都是静态页面。这就需要程序员的配合将页面在反馈给客户端之前将页面全部转换成静态页面。好了基本架构就这样，下面说说我怎么猜到的以及具体的架构：

法宝之一： **nslookup**

实战：

nslookup www.sina.com.cn

Server: ns-px.online.sh.cn

Address: 202.96.209.5

Non-authoritative answer:

Name: taurus.sina.com.cn

**Addresses: 61.172.201.230, 61.172.201.231, 61.172.201.232, 61.172.201.233
61.172.201.221, 61.172.201.222, 61.172.201.223, 61.172.201.224,
61.172.201.225**

61.172.201.226, 61.172.201.227, 61.172.201.228, 61.172.201.229

Aliases: www.sina.com.cn, jupiter.sina.com.cn

这里可以看到新浪在首页上用到了那么多 **IP**，开始有人会想果然新浪财大气粗啊。其实不然，继续往下看：

nslookup news.sina.com.cn

Server: ns-px.online.sh.cn

Address: 202.96.209.5

Non-authoritative answer:

Name: taurus.sina.com.cn

**Addresses: 61.172.201.228, 61.172.201.229, 61.172.201.230, 61.172.201.231
61.172.201.232, 61.172.201.233, 61.172.201.221, 61.172.201.222,
61.172.201.223**

61.172.201.224, 61.172.201.225, 61.172.201.226, 61.172.201.227

Aliases: news.sina.com.cn, jupiter.sina.com.cn

细心的人可以发现了 **news** 这个频道的 **ip** 数和首页上一样，而且 **IP** 也完全一样。也就是这些 **IP** 在 **sina** 的 **DNS** 上的名字都叫 **taurus.sina.com.cn**，那些 **IP** 都是这个域的 **A** 记录。而 **news,sports,jczs.news**。。。都是 **CNAME** 记录。用 **DNS** 来做自动轮询。还不信，再来一个，就体育频道好了：

nslookup sports.sina.com.cn

Server: ns-px.online.sh.cn

Address: 202.96.209.5

Non-authoritative answer:

Name: taurus.sina.com.cn

**Addresses: 61.172.201.222, 61.172.201.223, 61.172.201.224, 61.172.201.225
61.172.201.226, 61.172.201.227, 61.172.201.228, 61.172.201.229,**

61.172.201.230

61.172.201.231, 61.172.201.232, 61.172.201.233, 61.172.201.221

Aliases: sports.sina.com.cn, jupiter.sina.com.cn

其他的可以自己试。好了再来看看 *sohu* 的情况:

nslookup www.sohu.com

Server: ns-px.online.sh.cn

Address: 202.96.209.5

Non-authoritative answer:

Name: pagegrp1.sohu.com

Addresses: 61.135.132.172, 61.135.132.173, 61.135.132.176, 61.135.133.109

61.135.145.47, 61.135.150.65, 61.135.150.67, 61.135.150.69, 61.135.150.74

61.135.150.75, 61.135.150.145, 61.135.131.73, 61.135.131.91, 61.135.131.180

61.135.131.182, 61.135.131.183, 61.135.132.65, 61.135.132.80

Aliases: www.sohu.com

nslookup news.sohu.com

Server: ns-px.online.sh.cn

Address: 202.96.209.5

Non-authoritative answer:

Name: pagegrp1.sohu.com

Addresses: 61.135.150.145, 61.135.131.73, 61.135.131.91, 61.135.131.180

61.135.131.182, 61.135.131.183, 61.135.132.65, 61.135.132.80,

61.135.132.172

61.135.132.173, 61.135.132.176, 61.135.133.109, 61.135.145.47,

61.135.150.65

61.135.150.67, 61.135.150.69, 61.135.150.74, 61.135.150.75

Aliases: news.sohu.com

情况和 *sina* 一样, 只是从表面来看 *sohu* 的 *IP* 数要多于 *sina* 的 *IP* 数, 那么 *sohu* 上各个频道用的服务器就要多于 *sina* 了? 当然不能这么说, 因为一台服务器可以绑定多个 *IP*, 因此不能从 *IP* 数的多少来判断用了多少服务器。

从上面这些实验可以基本看出 *sina* 和 *sohu* 对于频道等栏目都用了相同的技术, 即 *squid* 来监听这些 *IP* 的 80 端口, 而真正的 *web server* 来监听另外一个端口。从用户的感觉上来说不会有任何的区别, 而相对于将 *web server* 直接和客户端连在一起的方式, 这样的方式明显的节省的带宽和服务器。用户访问的速度感觉也会更快。

先说那么多了, 要去睡觉了, 明天还有很多工作要做~有不明白的记得给我留言!!!

I 原 中国顶级门户网站架构分析 2

中国顶级门户网站架构分析 1

前天讲了最基本的推测方法，今天稍微深入一些：)

1. 难道就根据几个域名的 **ip** 相同就可以证明他们是使用 **squid** 的嘛？

当然不是，前面都只是推测。下面才是真正的证实我上面的猜测。先 **nslookup** 一把 **sina** 的体育频道。

```
nslookup sports.sina.com.cn
```

```
Server: ns1.china.com
```

```
Address: 61.151.243.136
```

```
Non-authoritative answer:
```

```
Name: taurus.sina.com.cn
```

```
Addresses: 61.172.201.231, 61.172.201.232, 61.172.201.233, 61.172.201.9
```

```
61.172.201.10, 61.172.201.11, 61.172.201.12, 61.172.201.13,
```

```
61.172.201.14
```

```
61.172.201.15, 61.172.201.16, 61.172.201.17, 61.172.201.227,
```

```
61.172.201.228
```

```
61.172.201.229, 61.172.201.230
```

```
Aliases: sports.sina.com.cn, jupiter.sina.com.cn
```

然后直接访问这些 **ip** 中的任意一个 **ip** 试试看，访问下来的结果应该是如下图所示：

由此可以证明 **sina** 是在 **dns** 中设置了很多 **ip** 来指向域名 **sqsh-19.sina.com.cn**，而其他各种相同性质的频道都只是 **sqsh-19.sina.com.cn** 一个别名，用 **CNAME** 指定。**dns** 的设置应该是这样的，然后 **server** 方面，通过 **squid 2.5.STABLE5**（最新的稳定版为 **STABLE6**）来侦听 **80** 端口。上面这些是根据一些信息分析而出的，应该基本正确的。下面一些就是我的个人的猜想：

它的真正的 **web server** 也同样是侦听 **80** 端口，因为在 **squid** 配置文件中有一项是：

```
httpd_accel_port 80
```

如果你设成其他端口号（比如 **88**）的话，那上图的错误信息就会变成

While trying to retrieve the URL: <http://61.172.201.19:88>

工具 2: **nmap** 扫描程序：可以用来检查服务器开了什么端口。

我现在用 **nmap** 来扫描 **sina** 的一个 **ip**: **61.172.201.19** 来进行分析

```
bash-2.05$ nmap 61.172.201.19
```

```
Starting nmap 3.50 ( http://www.insecure.org/nmap/ ) at 2004-07-30 13:31 GMT
```

```
Interesting ports on 61.172.201.19:
```

```
(The 1657 ports scanned but not shown below are in state: filtered)
```

```
PORT STATE SERVICE
```

```
22/tcp open ssh
```

```
80/tcp open http
```

```
Nmap run completed -- 1 IP address (1 host up) scanned in 73.191 seconds
```


可以看到他对外只开了 2 个端口，80 端口就是刚才我们说的 **squid** 打开的，这点刚才已经验证过了。而 22 端口是用来 **ssh** 远程连接的，主要是 **sa** 用来远程操作服务器用的安全性非常高的方法。

工具 3: lynx 或者其他可以读取 **http** 头文件的工具及小程序：直接看例子比较好理解：)

```
HTTP/1.0 200 OK
Date: Fri, 30 Jul 2004 05:49:47 GMT
Server: Apache/2.0.49 (Unix)
Last-Modified: Fri, 30 Jul 2004 05:48:16 GMT
Accept-Ranges: bytes
Vary: Accept-Encoding
Cache-Control: max-age=60
Expires: Fri, 30 Jul 2004 05:50:47 GMT
Content-Length: 180747
Content-Type: text/html
Age: 37
X-Cache: HIT from sqsh-230.sina.com.cn
Connection: close
```

上面是 **sina** 的 **http** 头的反馈信息。里面有很多有价值的东东哦：) 譬如，它后面的 **apache** 是用 **2.0.49**，还设了过期时间为 2 分钟。最后修改时间。这些都是要在编译 **apache** 的时候载入的，特别是 **Last-Modified** 还需要小小的改一把源码--至少我是这样做的。

综上所述

sina 的架构应该是前面 **squid**，按照现在的服务器 2u，2g 内存一般每台服务器至少可以跑 4 个 **squid2.5stable5**。这样它 16 个 ip 就用了 4 台服务器。后面一层是 **apache2.0.49** 应该会用 2 台。这 2 台可能用的全是私有 ip，通过前面的 **squid** 服务器在 **hosts** 文件中指定。具体的实现方法我会下次整理出我做实验的文档：) 而 **apache** 的 **htdocs** 可能是有一个或 2 个磁盘阵列作 **nfs**。**apache mount nfs server** 的时候应该是只读的，然后另外还有服务器专门用来做编辑器服务器，用来编辑人员更新文章。这台服务器应该对 **nfs server** 是具有可写的权限。

----这就一套完整的 **sina** 所运用的方案，当然很多是靠猜测的，我没有和 **sina** 的技术人员有过任何沟通（因为一个也不认识），否则我也就不会写出来了。其他 **sohu**，**163** 应该也有这样的架构。

最后声明：这只是一些静态页面组成频道的一个架构，**sina** 还有很多其他服务器，什么下载，在线更新等不在这个架构中。

I 服务器的大用户量的承载方案

http://blog.chinaunix.net/u/243/showart_299315.html

一、前言

二、编译安装

三、安装 **MySQL**、**memcache**

四、 安装 Apache、PHP、eAccelerator、php-memcache

五、 安装 Squid

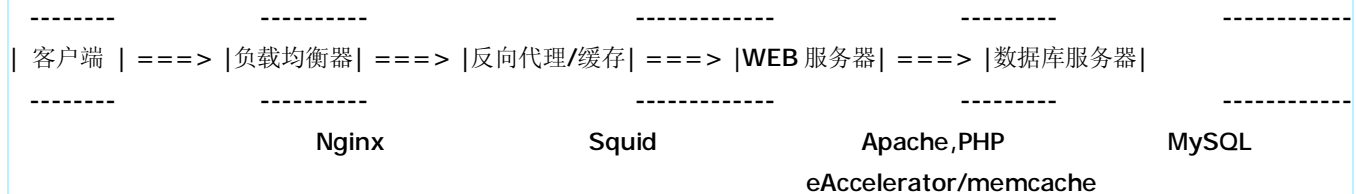
六、 后记

一、 前言

一、 前言，准备工作

当前，**LAMP** 开发模式是 **WEB** 开发的首选，如何搭建一个高效、可靠、稳定的 **WEB** 服务器一直是个热门主题，本文就是这个主题的一次尝试。

我们采用的架构图如下：



准备工作：

服务器： Intel(R) Xeon(TM) CPU 3.00GHz * 2, 2GB mem, SCISC 硬盘

操作系统： CentOs4.4, 内核版本 2.6.9-22.ELsmp, gcc 版本 3.4.4

软件：

Apache 2.2.3（能使用 MPM 模式）

PHP 5.2.0（选用该版本是因为 5.2.0 的引擎相对更高效）

eAccelerator 0.9.5（加速 PHP 引擎，同时也可以加密 PHP 源程序）

memcache 1.2.0（用于高速缓存常用数据）

libevent 1.2a（memcache 工作机制所需）

MySQL 5.0.27（选用二进制版本，省去编译工作）

Nginx 0.5.4（用做负载均衡器）

squid-2.6.STABLE6（做反向代理的同时提供专业缓存功能）

I YouTube Scalability Talk

Cuong Do of YouTube / Google recently gave a [Google Tech Talk on scalability](#).

I found it interesting in light of my own [comments on YouTube's 45 TB](#) a while back.

Here are my notes from his talk, a mix of what he said and my commentary:

In the summer of 2006, they grew from 30 million pages per day to 100 million pages per day, in a 4 month period. (Wow! In most organizations, it takes nearly 4 months to pick out, order, install, and set up a few servers.)

YouTube uses Apache for FastCGI serving. (I wonder if things would have been easier for them had they chosen [nginx](#), which is apparently wonderful for FastCGI and less problematic than Lighttpd)

YouTube is coded mostly in Python. Why? “Development speed critical”.

They use [psyco](#), Python -> C compiler, and also C extensions, for performance critical work.

They use [Lighttpd](#) for serving the video itself, for a big improvement over Apache.

Each video hosted by a “mini cluster”, which is a set of machines with the same content. This is a simple way to provide headroom (slack), so that a machine can be taken down for maintenance (or can fail) without affecting users. It also provides a form of backup.

The most popular videos are on a CDN (Content Distribution Network) - they use external CDNs and well as Google’s CDN. Requests to their own machines are therefore tail-heavy (in the “Long Tail” sense), because the head codes to the CDN instead.

Because of the tail-heavy load, random disks seeks are especially important (perhaps more important than caching?).

YouTube uses simple, cheap, commodity Hardware. The more expensive the hardware, the more expensive everything else gets (support, etc.). Maintenance is mostly done with rsync, SSH, other simple, common tools.

The fun is not over: Cuong showed a recent email titled “3 days of video storage left”. There is constant work to keep up with the growth.

Thumbnails turn out to be surprisingly hard to serve efficiently. Because there, on average, 4 thumbnails per video and many thumbnails per pages, the overall number of thumbnails per second is enormous. They use a separate group of machines to serve thumbnails, with extensive caching and OS tuning specific to this load.

YouTube was bit by a “too many files in one dir” limit: at one point they could accept no more uploads (!!) because of this. The first fix was the usual one: split the files across many directories, and switch to another file system better suited for many small files.

Cuong joked about “The Windows approach of scaling: restart everything”

Lighttpd turned out to be poor for serving the thumbnails, because its main loop is a bottleneck to load files from disk; they addressed this by modifying Lighttpd to add worker threads to read from disk. This was good but still not good enough, with one thumbnail per file, because the enormous number of files was terribly slow to work with (imagine tarring up many million files).

Their new solution for thumbnails is to use Google’s [BigTable](#), which provides high performance for a large number of rows, fault tolerance, caching, etc. This is a nice (and rare?) example of actual synergy in an acquisition.

YouTube uses MySQL to store metadata. Early on they hit a Linux kernel issue which prioritized the page cache higher than app data, it swapped out the app data, totally overwhelming the system. They recovered from this by removing the swap partition (while live!). This worked.

YouTube uses [Memcached](#).

To scale out the database, they first used MySQL replication. Like everyone else that goes down this path, they eventually reach a point where replicating the writes to all the DBs, uses up all the capacity of the slaves. They also hit a issue with threading and replication, which they worked around with a very clever “cache primer thread” working a second or so ahead of the replication thread, prefetching the data it would need.

As the replicate-one-DB approach faltered, they resorted to various desperate measures, such as splitting the video watching in to a separate set of replicas, intentionally allowing the non-video-serving parts of YouTube to perform badly so as to focus on serving videos.

Their initial MySQL DB server configuration had 10 disks in a RAID10. This does not work very well, because the DB/OS can't take advantage of the multiple disks in parallel. They moved to a set of RAID1s, appended together. In my experience, this is better, but still not great. An approach that usually works even better is to intentionally split different data on to different RAIDs: for example, a RAID for the OS / application, a RAID for the DB logs, one or more RAIDs for the DB table (uses “tablespaces” to get your #1 busiest table on separate spindles from your #2 busiest table), one or more RAID for index, etc. Big-iron Oracle installation sometimes take this approach to extremes; the same thing can be done with free DBs on free OSs also.

In spite of all these effort, they reached a point where replication of one large DB was no longer able to keep up. Like everyone else, they figured out that the solution database partitioning in to “shards”. This spread reads and writes in to many different databases (on different servers) that are not all running each other's writes. The result is a large performance boost, better cache locality, etc. YouTube reduced their total DB hardware by 30% in the process.

It is important to divide users across shards by a controllable lookup mechanism, not only by a hash of the username/ID/whatever, so that you can rebalance shards incrementally.

An interesting DMCA issue: YouTube complies with takedown requests; but sometimes the videos are cached way out on the “edge” of the network (their caches, and other people's caches), so its hard to get a video to disappear globally right away. This sometimes angers content owners.

Early on, YouTube leased their hardware.

I High Performance Web Sites by Nate Koechley

One dozen rules for faster pages

1. Share results of our research in t Yahoo is firmI committed to openness

Why talk about performance?

In the last 2 years, we do a lot more with web pages. Steve Souders - High Performance Two Performance Flavors: Response Time and System Efficiency The importance of front end performance!!! 95% is front end. Backend vs. Front-end Until now we have on 1 Perception - How fast does it feel to the users? Perceived response time It's in the eye of the beholder 2 80% of consequences Yahoo Interface Blog yuiblog.com 3 Cache Sadly the cache doesn't work as well as it should 40-60% of users still have an empty cache Therefore optimize for no-cache and with cache 4 Cookies Set scope correctly Keep sizes low, 80ms delay with cookies Total cookie size - Amazon 60 bytes - good example. 1. Eliminate unnecessary cookies 2. Keep cookie sizes low 3. 5

Parallel Downloads

One Dozen Rules

Rule 1 - Make fewer HTTP requests [css sprites alistapart.com/articles/sprites](http://css.sprites.alistapart.com/articles/sprites) Combine Scripts, Combined Stylesheets Rule 2 - Use a CDN amazon.com - Akamai Distribute your static content before distributing content Rule 3 Add an Expires Header Not just for images images, stylesheets and scripts Rule 4: Gzip Components You can add to users download times 90% of browsers support compression Gzip compresses more than deflate Gzip: not just for HTML for gzip scripts, Free YUI Hosting includes Aggregated files w Rule 5: Put CSS at the top stylesheets use `< link >` not `@import!!!!` Slower, but perceived loading time is faster Rule 6; Move scripts to the bottom of the page scripts block rendering what about defer? - no good Rule 7: Avoid CSS Expressions Rule 8: Make JS and CSS External Inline: bigger HTML but no http request External: cachable but extra http Except for a users "home page" Post-Onload Download Dynamic Inlining Rule 9: Reduce DNS Lookups Best practice: Max 2-4 hosts Use keep-alive Rule 10: Minify Javascript Take out white space, Two popular choices - Dojo is a better compressor but JSMIn is less error prone. minify is safer than obfuscation Rule 11: Avoid redirects Redirects are worst form of blocking Redirects - Amazon have none! Rule 12: Turn off ETags

Case Studies

Yahoo 1 Moved JS to onload 2 removed redirects 50% faster What about performance and Web 2.0 apps? Client-side CPU is more of an issue User expectations are higher start off on the right foot - care! Live Analysis IBM Page Detailer - windows only Fasterfox - measures load time of pages LiveHTTPHeaders firefox extension Firebug - Recommended! YSlow to be released soon.

Conclusion

Focus on the front end harvest the low hanging fruit reduce http requests

I Rules for High Performance Web Sites

These rules are the key to speeding up your web pages. They've been tested on some of the most popular sites on the Internet and have successfully reduced the

response times of those pages by 25-50%.

The key insight behind these best practices is the realization that only 10-20% of the total end-user response time is spent getting the HTML document. You need to focus on the other 80-90% if you want to make your pages noticeably faster. These rules are the best practices for optimizing the way servers and browsers handle that 80-90% of the user experience.

- Rule 1 - Make Fewer HTTP Requests
- Rule 2 - Use a Content Delivery Network
- Rule 3 - Add an Expires Header
- Rule 4 - Gzip Components
- Rule 5 - Put CSS at the Top
- Rule 6 - Move Scripts to the Bottom
- Rule 7 - Avoid CSS Expressions
- Rule 8 - Make JavaScript and CSS External
- Rule 9 - Reduce DNS Lookups
- Rule 10 - Minify JavaScript
- Rule 11 - Avoid Redirects
- Rule 12 - Remove Duplicate Scripts
- Rule 13 - Turn Off ETags
- Rule 14 - Make AJAX Cacheable and Small

I 对于应用高并发，DB 千万级数量该如何设计系统哪？

背景：

博客类型的应用，系统实时交互性比较强。各种统计，计数器，页面的相关查询之类的都要频繁操作数据库。数据量要求在千万级，同时在线用户可能会有几万人活跃。系统现在是基于 spring + hibernate + jstl + mysql 的，在 2 千人在线，几十万记录下没有什么压力。可对于千万记录以及数万活跃用户没什么经验和信心。

对于这些，我的一点设计想法与问题，欢迎大家指导：

一. 加强 cache

由于 web2 类型的网站，用 squid 反向代理可能不是很适用；由于这种情况下需要 cluster，jvm 上作过多 cache 可能会引起其他问题；所以比较合适的应该是采用静态发布的方式，把数据发布成 xml 文件，然后通过 xml + xslt 拼接各模块(div)显示。（直接发布成 html 文件用 jstl 感觉不是很方便，也没用过，请有经验的介绍下），主要目的是把压力拦截在 Apache 上。或者用 memcached cache 文章内容，用户资料等对象。

二. 数据库分库

分库有两种，一种是分表，把经常访问的放一张表，不常访问的放一张表。

好比对于博客，文章表可以分为文章基本信息（标题，作者，正文……）不常改动的信息，和文章统计信息（阅读次数，评论次数……）经常变动的信息，以期望 update 统计信息之类的可以快一点（这个东西实践起来弊端也会比较明显：查询文章时需要多查询一次统计信息表，到底能不能提高性能还没有具体数据，欢迎有经验的给点数据：））。

对于记录过多，好比千万级，这样的分法显然也解决不了问题，那么就需要归档处理了。归档大致就是创建一个同样的表，把旧内容（好比三个月以前的）都移到旧表里面，保持活跃的表记录不多。（mysql 本身有一个 archive 引擎，看资料感觉对解决大量数据没什么用处，连索引都不支持，用过的朋友可以给点建议）。归档带来的最大问题就是：归档以后的数据如何访问哪？如果用户要访问以前的数据就会比较麻烦了。（mysql 的 merge 查询？）大家这方面有没有好的 practice？我还没想到好的办法。

分库的另外一种方式是物理的分，就是装他几十台 mysql 服务器，然后按照某种方式把数据分散到不同的服务器上，这种方式也有利于备份恢复和系统的稳定性（一台数据库宕了，也只会影响一部分功能或用户）。例如对于博客应用，比较理想的分库模式可以按照用户分，好比我把用户 id 在 1…10 万的材料都存到 mysql 1 上，把 10 万。。。20 万的存到 mysql 2…。依次类推，通过线性增加服务器的方式解决大数据问题。呵呵，还算完美吧~~，就是给统计排名带来了麻烦……

按照第二种分库方式，数据库连接将发生变化，如果数据达到千万，10 几个 mysql 应该是需要的，这时候连接池就要废掉了，采用每次查询取链接的方式。或者需要改造出一个特别的连接池了。

三. 采用 Ibatis

把 hibernate 废掉，改用 ibatis，毕竟 ibatis 可以很方便的进行 sql 优化，有什么问题优化起来方便多了（还没有用过 ibatis，只是感觉）。另一方面，如果物理分库有效果，好像严格的 sql 优化意义也就不大了。这应该也是一个优化方面。

总结一下我的结构：把文章，用户资料，各种分类，tag，链接，好友之类的进行静态化（xml + xslt 读取显示）+ 物理分库 + ibatis sql 优化 + JVM 短暂性的 cache 总的用户数，在线用户数等极个别数据，其他的全部不 cache（包括关闭 hibernate 二级缓存，如果用 hibernate）

各个博客之间没什么关系，采用分库+分表的方法应该比较好的。

都不用按常用不常用分，简单地将博客分组就好了。

另外，因为业务逻辑比较简单，要处理千万记录以及数万活跃用户，我觉得还是用 JDBC+mysql，自己从头构建一个应用服务器更好些。。

MySQL 5.1 已经支持表分区了，拿 100 万行的表测试过（采用的是 HASH），查询速度非常理想。

可以使用 velocity 模板，直接发布为 html。

另外，je2 里面有很多静态页面，不知道是如何自动生成的。看起来效果不错。

<http://www.javaeye.com/static.html>

我以前用 `httpClient` 读页面，然后写成本地文件，少量的效果还不错（速度和静态页面效果都不错）。项目要定时循环读，结果后来因为任务多了 `quartz` 调度不了那么快，造成内存溢出。

没用过 `java`，表是肯定要分的，具体怎么分要看你们的具体应用需求，分表后对外的读取接口可以封装起来，内部处理数据定位问题。`cache` 尽量走内存少走文件，否则数量和访问量上去以后 `io` 也够受的。系统的几大模块间尽量独立，互相用消息队列异步通信。

skybyte

大并发系统设计

#1

中级会员



注册日期:

2006/9/3 17:42

所属群组:

会员

帖子: 62

等级: 6; EXP: 75

HP: 0 / 143

MP: 20 / 405



大并发系统设计

2007-2-15

杨思勇

Email: yangsy.cq (啊特) gmail (点) com

架构设计

优化服务器配置。
负载均衡技术。

加大内存。
加大并发数。
升级操作系统版本。
正确的磁盘分区技巧。

负载均衡技术

DNS 负载均衡技术。

优点：优点是经济简单易行，节点可以在任意位置。

缺点：更新慢，节点宕机后无法响应。

交换机负载均衡技术。

优点：能及时响应节点宕机，速度快。

缺点：对交换机有要求，节点必须在交换机中。

Web 容器采用线程池技术。

进程模式的请求响应非常慢。但是比较稳定，一个进程 **dead** 后不影响其它进程。

采用线程池技术的后响应速度非常快，数据可以在线程之间共享。缺点是有可能单个线程会影响其它线程，并且有可能会发生死锁。

数据库连接采用连接池技术

提高了响应时间，尤其是的 **SQL** 比较多时更应采用连接池技术。

注意：
连接的释放。
连接的事务处理。

页面预编译技术。

编译后的代码执行速度要比脚本语言高出几个数量级。
Jsp 主要是第一次运行时编译，这样可以提高第二次响应请求的时间。
可以在部署后批量编译所有动态需要编译的文件。

缓存设计技术。

缓存能大大减少数据量的压力。
页面全部缓存。
优点：整个页面响应速度快。
缺点：更新不及时，无法单独刷新某一块。
单个组件缓存。
优点：执行速度快，可以很细的控制需要缓存的部分，节省内存空间。
优秀的缓存方案
页面级缓存技术有：squid、OSCache 的 taglib 技术等
组件级有：MEMCache、OSCache、ECache 等

高度优化 SQL、索引、分页

值采用?形式来复用 SQL，如：insert into table(f1,f2) values (?,?)。数据库会缓存这些 sql，不会再解析了。

关联表查询注意要使用到索引。最好通过他表的主键关联。
采用存贮过程技术。
时间存贮采用时间类型，数据库对 `date` 类型字段都做了优化。
经常要查询的字段必须建索引，使用到的索引上最好能排除全表的 80% 的记录。
如果不能做到，则需要建立联合索引。同样索引必须能排除 80% 以上的记录。
索引定期优化，重建。
查询排序最好通过主键来排序。
一张表上索引不要超过 5 个。
尽量不要 `Like` 查询大字段。
执行时间超过 100ms 的 SQL 基本上都有问题，要么是设计的问题，要么是 SQL 没有优化，要么是索引没有使用正确。

数据流压缩技术

数据流压缩主要用在 web 服务器和浏览器之间的数据传送。
现在的浏览器基本上都支持 `gzip` 和 `deflate` 压缩技术。
注意压缩比。
不要压缩 `jpg`、`rar`、`zip` 等已经压缩过的文件。否则性能会更低。

I 高性能服务器设计

http://blog.chinaunix.net/u/5251/showart_236329.html

书接上文，很自然地就到了高性能服务器设计这个话题上来了。

先后查看了 `haproxy`、`l7sw` 和 `lighttpd` 的相关源码，无一例外，他们一致认为多路复用是性能最好的服务器架构。事实也确实应该如此，进程的出现一方面就是为了保存任务的执行上下文从而简化应用程序设计，如果程序的逻辑结构不是很复杂，那么用整个进程控制块来保存执行上下文未免有些大材小用，加上进程调度和其他的一些额外开销，程序设计上的高效很可能会被执行时的低效所抵消。代价也是有的：程序设计工作将更加具有挑战性。

体系结构选定之后，我们就要考虑更加细节的部分，比如说用什么操作系统，用操作系统提供的那些 **API**。在这方面，前辈们已经做过很多，我们只需要简单的“拿来”即可，如果再去枉费唇舌，简直就是浪费时间，图财害命。**High-Performance Server Architecture** 从根本上分析了导致服务器低效的罪魁祸首：数据拷贝、（用户和内核）上下文切换、内存申请（管理）和锁竞争；**The C10K Problem** 列举并分析了 **UNIX**、**Linux** 甚至是部分 **Windows** 为提高服务器性能而设计的一些系统调用接口，这篇文档的难能可贵之处还在于它一致保持更新；**Benchmarking BSD and Linux** 更是通过实测数据用图表的形式

把 BSD 和 Linux 的相关系统调用的性能直观地陈列在我们眼前，结果还是令人激动的：Linux 2.6 的相关系统调用的时间复杂度竟然是 $O(1)$ 。

简单的总结如下：

1. 操作系统采用 Linux 2.6.x 内核，不仅因为它的高性能，更因为它大开源（这并不是说其他的 UNIX 或者是 BSD 衍生物不开源）给程序设计带来的便利，我们甚至可以把服务做到内核空间。
2. 多路复用采用 epoll 的“电平触发”(Level Triggered)模式，必要时可以采用“边缘触发”(Edge Triggered)，但要注意防止数据停滞。
3. 为避免数据拷贝可以采用 sendfile 系统调用发送小文件，或者是文件的小部分，注意避免 sendfile 因磁盘 IO 而导致的阻塞。
4. 如果服务操作设计大量磁盘 IO 操作，应选用 Linux 内核提供的异步 IO 机制，其对应的用户空间库为 libaio，注意：这里提到异步 IO 库并非目前 glibc 中附带的异步 IO 实现。
5. 如果同时有多个数据需要传输，采用 writev/readv 来减少系统调用所带来的上下文切换开销，如果数据要写到网络套接字文件描述符，这也能在一定程度上防止网络上出现比较小帧，为此，还可以有选择地开启 TCP_CORK 选项。
6. 实现自己的内存管理，比如说缓存数据，复用常用数据结构等。
7. 用多线程替代多进程，线程库当然选择 nptl。
8. 避免进程/线程间非必要的同步，保持互斥区的短小。

上面这些琐碎的细节在 ESR 看来可能都是过早优化，他可能又会建议我们等待硬件的升级。哈哈，提醒还是不无道理的，算法的设计部分，我们更要下大力气，因地制宜地降低算法的时间复杂度。为什么不提空间复杂度呢？内存的价格还是相对低廉吧，不过还是不要忘记现在的计算机瓶颈多在内存的访问。

有一点需要提醒一下，目前 SMP 系统和多核心 CPU 比较常见，如果还是仅采用单进程（线程）的多路复用模型，那么同一时间将只有一个 CPU 为这个进程（线程）服务，并不能充分发挥 CPU 的计算能力，所以需要至少 CPU（CPU 核心）数目个进程（线程）来分担系统负担。有一个变通的解决方案：不用修改源码，在服务器上运行两个服务程序的实例，当然这个时候服务端口应该是不同的，然后在其前端放置负载均衡器将流量和连接平均分配到两个服务端口，可以简单的通过 DNAT 来实现负载均衡。其实，这个时候我们已经把多 CPU 或者是多核系统看成了多个系统组成的集群。

为了提高服务器的性能，单纯的依靠提高单个服务器的处理能力似乎不能奏效，况且配置越高的服务器花销也就越高，为此人们经常采用服务器集群的方式，通过把计算尽可能地分配到相对比较廉价的机器上单独完成，籍此来提升服务器的整体性能，事实证明，这种体系结构不仅是切实可行的，而且还能提高服务器的可用性，容错能力也较强。在网络服务器方面，Linux 内核中的由国人章文嵩先生设计的 IP 层负载均衡解决方案 LVS 比较有名，还有就是工作于应用层的 haproxy 和刚刚起步的 l7sw。

I 优势与应用：再谈 CDN 镜像加速技术

来源：中国 IDC 圈 时间：2007-1-22 作者：佚名 保存本文 [进入论坛](#)

CDN，全称是 Content Delivery Network，中文可译为“内容快递网”。它是一个建立并覆盖在互联网（Internet）之上的一层特殊网络，专门用于通过互联网高效传递丰富的多媒体内容。CDN 出现和存在的意义在于它使互联网更有效地为人们服务，特别是那些对互联网内容有更高要求（比如由简单的文字和图片等静态内容到声像俱全的多媒体动态内容）的人们。

“CDN 技术”简介

CDN 的全称是 Content Delivery Network，即内容分发网络。其目的是通过在现有的 Internet 中增加一层新的网络架构，将网站的内容发布到最接近用户的网络“边缘”，使用户可以就近取得所需的内容，解决 Internet 网络拥挤的状况，提高用户访问网站的响应速度。从技术上全面解决由于网络带宽小、用户访问量大、网点分布不均等原因所造成的用户访问网站响应速度慢的问题。

目前，国内访问量较高的大型网站如新浪、网易等，均使用 CDN 网络加速技术，虽然网站的访问巨大，但无论在什么地方访问都会感觉速度很快。而一般的网站如果服务器在网通，电信用户访问很慢，如果服务器在电信，网通用户访问又很慢。

“CDN 技术”的优势

- 1、本地 Cache 加速 提高了企业站点（尤其含有大量图片和静态页面站点）的访问速度，并大大提高以上性质站点的稳定性
- 2、镜像服务 消除了不同运营商之间互联的瓶颈造成的影响，实现了跨运营商的网络加速，保证不同网络中的用户都能得到良好的访问质量。
- 3、远程加速 远程访问用户根据 DNS 负载均衡技术 智能自动选择 Cache 服务器，选择最快的 Cache 服务器，加快远程访问的速度
- 4、带宽优化 自动生成服务器的远程 Mirror（镜像）cache 服务器，远程用户访问时从 cache 服务器上读取数据，减少远程访问的带宽、分担网络流量、减轻原站点 WEB 服务器负载等功能。
- 5、集群抗攻击 广泛分布的 CDN 节点加上节点之间的智能冗余机制，可以有效地预防黑客入侵以及降低各种 D.D.o.S 攻击对网站的影响，同时保证较好的服务质量。

网站用“CDN 技术”武装的流程

第一步：修改 DNS 解析

前面已经说到，CDN 其实是夹在网页浏览者和被访问的服务器中间的一层镜像或者

说缓存，浏览者访问时点击的还是服务器原来的 URL 地址，但是他看到的内容其实是离他的 IP 地址所在地最近的一台镜像服务器上的页面缓存内容，也就是说用户在使用原来的 URL 访问服务器时并没有实际访问到服务器上的内容，所以要实现这个效果，就得在这个服务器的域名解析上进行一些调整。

实际上，这个服务器的域名解析过程已经转变为为访问者选择离他最近的镜像服务器，因此域名的解析服务器的 IP 要改成 CDN 运营商架设的智能解析服务器的 IP，例如你在新网注册一个域名，默认用的就是新网的 DNS 服务器为你进行解析，而假设你选择了网宿的 CDN 服务，就得修改域名管理的设置，改成使用网宿的 CDN 解析服务器来进行解析。

这样，当一个浏览者访问你的网站时，他访问的 URL 地址就会被网宿的 CDN 解析服务器解析到网宿科技各地镜像服务器中离这个浏览者最近的一台上面。

第二步：调整网页架构

CDN 既然是一种缓存技术，那么它的实时性肯定是无法实现的，镜像服务器上的缓存一般都是隔一定的时间更新一次，因此在更新期间内，用户看到的内容是不会变的；所以使用 CDN 加速的服务器应该以静态页面和实时更新频率较低的内容为主，像论坛、天气预报这种内容更新频繁的站点使用 CDN 反而适得其反。

CDN 最适合的领域是资讯提供站点或者其他以静态页面为主的内容展示性质站点。

第三步：镜像服务器自动高新缓存

镜像服务器上面安装有一个可以进行自动远程备份的软件，当然它只备份静态页面和图片这些，每隔一定的时间，各个镜像服务器就会到网站的源服务器上去获取最新的内容。

那么有些网友就觉得，如果源服务器已经更新了但是缓存服务器还没更新，那该怎么办？这个问题其实并不存在，如果用户访问的是缓存服务器上也没有的页面，那么镜像服务器会先从源服务器上拿到这个页面的缓存然后再发送给访问者，如果用户访问的是动态页面，那么这个访问请求就会被提交到源服务器。

“CDN 技术”的应用和效果

CDN 对于门户性质资讯站点的加速效果还是非常明显的，以新浪为例：

新浪采用了 ChinaCache 做的 CDN 系统，ChinaCache 在全国分布了四十多个点，同时采用基于动态 DNS 分配的全球服务器负载均衡技术。

从新浪的站点结构可以看出：

>www.sina.com.cn

Server: UnKnown

Address: 192.168.1.254

Non-authoritative answer:

Name: libra.sina.com.cn

Addresses: 61.135.152.71, 61.135.152.72, 61.135.152.73, 61.135.152.74 61.135.152.75,
61.135.152.76, 61.135.153.181, 61.135.153.182, 61.135.53.183, 61.135.153.184,
61.135.152.65, 61.135.152.66, 61.135.152.67, 61.135.12.68, 61.135.152.69, 61.135.152.70

Aliases: www.sina.com.cn, jupiter.sina.com.cn

在北京地区 ChinaCache 将 www.sina.com.cn 的网址解析到 libra.sina.com.cn，然后 libra.sina.com.cn 做了 DNS 负载均衡，将 libra.sina.com.cn 解析到 61.135.152.71 等 16 个 ip 上，这 16 个 ip 分布在北京的多台前台缓存服务器上，使用 squid 做前台缓存。如果是在其它地区访问 www.sina.com.cn 可能解析到本地相应的服务器，例如 pavo.sina.com.cn，然后 pavo 又对应了很多做了 squid 的 ip。这样就实现了在不同地区访问自动转到最近的服务器访问，达到加快访问速度的效果。

我们再看一个新浪其它频道是指到那里的：

> news.sina.com.cn

Server: UnKnown

Address: 192.168.1.254

Non-authoritative answer:

Name: libra.sina.com.cn

Addresses: 61.135.152.65, 61.135.152.66, 61.135.152.67, 61.135.152.68 61.135.152.69,
61.135.152.70, 61.135.152.71, 61.135.152.72, 61.135.152.73 61.135.153.178, 61.135.153.179,
61.135.153.180, 61.135.153.181, 61.135.153.182 61.135.153.183, 61.135.153.184

Aliases: news.sina.com.cn, jupiter.sina.com.cn

可以看出，各个频道的前台缓存集群与 www.sina.com.cn 的前台缓存集群是相同的。

新浪使用 CDN 后效果也非常明显：

这是在笔者在广州 ping 新浪域名，被解析到华南这边的镜像服务器，反映速度快，稳定无丢包：

```
C:\Documents and Settings\Administrator>ping www.sina.com.cn

Pinging jupiter.sina.com.cn [218.30.66.101] with 32 bytes of data:

Reply from 218.30.66.101: bytes=32 time=38ms TTL=245
Reply from 218.30.66.101: bytes=32 time=38ms TTL=245
Reply from 218.30.66.101: bytes=32 time=44ms TTL=245
Reply from 218.30.66.101: bytes=32 time=37ms TTL=245
```

假如没使用 CDN，还是访问新浪在北京架设的服务器，不仅反应速度慢了好几倍，甚至还出现超时：

```
C:\Documents and Settings\Administrator>ping 61.135.152.71

Pinging 61.135.152.71 with 32 bytes of data:

Reply from 61.135.152.71: bytes=32 time=130ms TTL=51
Reply from 61.135.152.71: bytes=32 time=133ms TTL=51
Reply from 61.135.152.71: bytes=32 time=130ms TTL=51
Request timed out.
```

“CDN 技术”与“镜像站点”的区别

CDN 有别于镜像，因为它比镜像更智能，或者可以做这样一个比喻：CDN=更智能的镜像+缓存+流量导流。因而，CDN 可以明显提高 Internet 网络中信息流动的效率。从技术上全面解决由于网络带宽小、用户访问量大、网点分布不均等问题，提高用户访问网站的响应速度。

I 除了程序设计优化，zend+ eacc(memcached)外，有什么办法能提高服务器的负载能力呢？

发表时间: 2007-7-03 17:35 作者: cdexs 来源: PHPChina 开源社区门户

字体: 小 中 大 | 打印

看到豆瓣网单台 AMD 服务器,能支撑 5w 注册用户,我想他同时在线用户不会低于 5K,那么,在(PHP)系统设计、系统加速方面, DB 方面做怎样的优化才能充分利用系统资源,最大限度的提高系统负载能力呢。

是不是还有其他的办法呢？

我也来说两句 [查看全部评论](#) 相关评论

- **Snake.Zero (2007-7-03 17:37:31)**

squid 不可少

- **cdexs (2007-7-03 18:42:16)**

楼上，我想在多台 server 时，squid 比较用的上，当我只有 1~2 台服务器呢？

- **虾球桑 (2007-7-04 03:06:24)**

豆瓣的程序应该是他们自己写的吧，如果你的也是自己写的，把脚本好好优化可以提速不少，优良的代码结构能比操蛋的结构高出几成的效率

- **cdexs (2007-7-04 10:04:32)**

楼上，在程序设计时，针对性能和效率有那些技巧和注意点呢??

- **Snake.Zero (2007-7-04 10:35:38)**

统计不是个小问题，所以建议独立一台服务器专门做统计，系统优化方面我不是很足的经验，如果是多台服务器的话

可以 3: 2: 1 或者 3: 2: 2 的方式，3 台 squid，2 台 PORTAL，2 台 DB 读写分离
另外你也可以选择分离文件服务器，把那些静态的东西尽量不要让 APACHE 来完成

- **cdexs (2007-7-04 11:26:38)**

楼上的对，我准备将站内的图片从逻辑中分离出来(现在物理上还是一台 server)。

- **pigpluspower (2007-7-04 14:55:49)**

最实际的做法（单，多机通用）：

尽量油画你的代码！

若是劣质的代码，即使你有 Zend Platform 由能怎么样？

除了一些技术性的优化以外，以下这一点小细节可以帮到不少的忙：

- 1、免除多余的空格，如“if”后面那个括号里面的空格
- 2、在开发过后记得要去掉所有的注释，以提高效率
- 3、避免“双重循环结构”，比如两个不同条件判断，却要运行同一项处理，if 循环中尽量使用“&&”连接（一般没有人会去写“双重循环结构”，除非代码过于复杂）
- 4、若可能，干脆把“换行”去掉，“;”后面直接接代码

- **cdexs (2007-7-04 14:58:59)**

楼上的我没法说你的。。。。。

- **Snake.Zero (2007-7-04 16:24:38)**

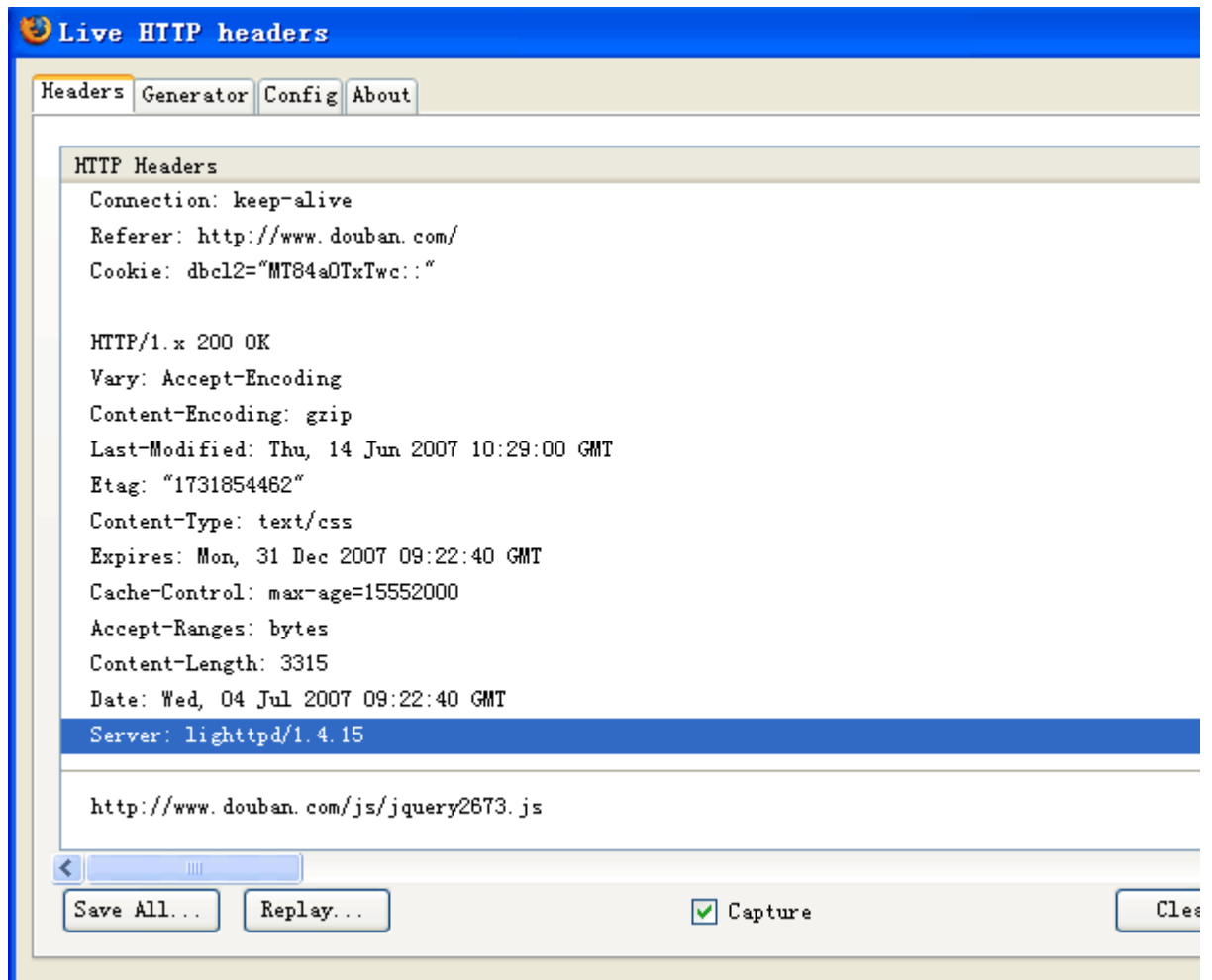
是啊，很寒

去注释是通过 php.exe 来完成的

另：楼主的代码编译过没？

- **太阳雨 (2007-7-04 17:28:05)**

豆瓣用的是: lighttpd/1.4.15



dou.png

- **cdexs (2007-7-04 20:08:07)**

用 eaccelerator,准备用 zend 编译过,再加上 eaccelerator。但看到有人说 eacc 对 zend 编过的代码不起任何作用,而且 zend optimizer 本身也要占用资源,所以一直在权衡.....大家给个建议哈

看到 lighhttpd + fastcgi 比 apache+mod_php 快,准备选用前种 web 服务。

看了下 xajax 的调用方法,想着,对于 ajax 的支持调用开销,xajax 是不是比 jquery 要小?比如有个情况,需要调用当前 php 页面中的用户是否存在的检测。用 jquery 可能需要:current.php?u=xxx。这个时候,页面接受参数,按照顺序执行到处理"u"参数(页面中的 chkusr()),再输出。但如果用 xajax,直接在 client 里调用后台页面方法 chkusr(arg),此时显然比 jquery 调用少了从页面开始第一条语句顺序执行而造成多余指令执行的资源浪费。偶没有仔细研究过 xajax 的实现细节。大家讨论讨论

[本帖最后由 cdexs 于 2007-7-4 20:17 编辑]

- **Snake.Zero (2007-7-04 21:25:52)**

lighhttpd 确实不错

- **ok7758521ok (2007-7-07 12:29:54)**



网站的拓扑结构

首先用户层

一般第一层采用 **cache** 拦截技术，（一般可以阻挡 50%以上的流量）

第二层应用层 --程序设计层

第三层数据缓存层层--memcache

第四层数据库（db）层

I 如何规划您的大型 JAVA 多并发服务器程序

文章作者：陈林茂 发布时间：2003 年 4-月 5 日 文章来源：转载 查看次数：643

版权申明：本站署名的原创文章，本站及作者享有版权，其他网站及传统媒体如需使用，转载时请注明出处和原作者。本站转载的文章如有侵犯到您的版权，请及时向本站提出。

JAVA 自从问世以来，越来越多的大型服务器程序都采用它进行开发，主要是看中它的稳定性及安全性，但对于一个新手来说，您又如何开发您的 **JAVA** 应用服务器，同时又如何规划您的 **JAVA** 服务器程序，并且很好的控制您的应用服务器开发的进度，最后，您又如何发布您的 **JAVA** 应用服务器呢？（由于很多前辈已有不错的著作，我只能在这里画画瓢，不足指出，请多来信指正，晚辈将虚心接受！本人的联系方式：

linmaochen@sohu.com）

废话少说，下面转入正题：

本文将分以下几个部分来阐述我的方法：

- 1、 怎样分析服务器的需求？
- 2、 怎样规划服务器的架构？
- 3、 怎样规划服务器的目录及命名规范、开发代号？
- 4、 原型的开发（-）： 怎样设计服务器的代码骨架？
- 5、 原型的开发（二）： 怎样测试您的代码骨架？
- 6、 详细的编码？
- 7、 如何发布您的 **JAVA** 服务器产品？

一、 如何分析服务器的需求？

我的观点是：

1. 服务器就像一台轧汁机，进去的是一根根的甘蔗，出来的是一杯杯的甘蔗汁；
也就是说，在开发服务器之前，先要明白，服务器的请求是什么？原始数据是什么？
接下来要弄明白，希望得到的结果是什么？ 结果数据应该怎样来表述？
其实要考虑的很多，无法一一列出（略）。

二、如何规划服务器的架构？

首先问大家一个小小的问题：在上海的大都市里，公路上的公交客车大致可以分为以下两类：
空调客车，票价一般为两块，上车不需要排队，能否坐上座位，就要看个人的综合能力；
无人售票车，票价一般 1 块和一块五毛，上车前需要规规矩矩排队，当然，座位是每个人都有的。
那么，我的问题是，哪类车的秩序好呢？而且上下车的速度快呢？答案是肯定的： 无人售票车。

所以，我一般设计服务器的架构主要为：

首先需要有一个请求队列，负责接收客户端的请求，同时它也应有一个请求处理机制，说到实际

上，应有一个处理的接口；

其次应该有一个输出队列，负责收集已处理好的请求，并准备好对应的回答；当然，它也有一个

回答机制，即如何将结果信息发送给客户端；

大家都知道，服务器程序没有日志是不行的，那么，服务器同时需要有一个日志队列，负责整个服

务器的日志信息收集和处理；

最后说一点，上公交车是需要有钞票的，所以，服务器同样需要有一个验证机制。

...(要说的东西实在太多，只好略)

三、 怎样规划服务器的目录及命名规范、开发代号

对于一般的大型服务器程序，应该有下面几个目录：

bin ： 主要存放服务器的可执行二进制文件；

common: 存放 JAVA 程序执行需要的支持类库；

conf ： 存放服务器程序的配置文件信息；

logs ： 存放服务器的日志信息；

temp ： 存放服务器运行当中产生的一些临时文件信息；

cache ： 存放服务器运行当中产生的一些缓冲文件；

src ： 当然是存放服务器的 JAVA 源程序啦。

.....（其他的设定，根据具体需求。）

四、原型的开发（-）： 怎样设计服务器的代码骨架？

1. 首先服务器程序需要有一个启动类，我们不妨以服务器的名字命名：(ServerName).class

2. 服务器需要有一个掌控全局的线程，姑且以：(MainThread.class)命名；

3. 注意不论是短连接和长连接，每一个客户端需要有一个线程给看着，以 ClientThread.class 命名

4. 请求队列同样需要以线程的方式来表现： (InputQuene.Class),对应的线程处理类以 InputProcessThread.class

命名；

5. 输出队列也需要一个线程： (OutputQuene.Class) ,对应的处理机制以 OutputProcessThread.class 命名；

6. 日志队列也是需要有一个线程的，我们以 logQuene.class,logQueneThread.Class 来命名；

7. 缓冲区的清理同样需要定时工作的，我们以 CacheThread.Class 来命名；

8. 如果您的参数信息是以 XML 的方式来表达的话，那么我也建议用一个单独的类来管理这些参数信息：

Config.Class

9. 当然，如果您想做得更细一点的话，不妨将客户端客服务器端的通讯部分也以接口的形式做出来：

CommInterface.Class

.....(太多，只能有空再说！)

五、 原型的开发（二）： 怎样测试您的代码骨架？

下面为原型的骨架代码，希望大家多多提点意见！谢啦！

/* 服务器描述：服务器主控线程

1. 读取组态文件信息

2. 建立需求输入队列

3. 建立需求处理输出队列

4. 建立需求处理线程

5. 建立输出预处理线程，进行需求处理结果的预处理

6. 建立缓冲区管理线程，开始对缓冲取进行管理

7. 建立服务连接套捷字，接受客户的连接请求，并建立客户连接处理线程

*/

```
import java.io.*;
```

```
import java.net.*;
```

```
import java.util.*;
```

```
public class mainThread extends Thread {
```

```

private ServerSocket serverSocket=null;

/*当前服务器监听的端口*/
private int serverPort;

public mainThread(String ConfUrl) {
    try{

        /*建立服务器监听套接字*/
        this.serverSocket =new ServerSocket(serverPort);

    }catch(Exception e){
        //
        System.out.println(e.getMessage());
    }
}

/*线程的执行绪*/
public synchronized void run(){

    while(listening){
        try{

            Socket sersocket =this.serverSocket.accept();

            ClientThread _clientThread=
                new ClientThread([ParamList]);

            _clientThread.start();

        }catch(Exception e){

        }

    }

    /*退出系统*/
    System.exit(0);
}

/*
1. 完成客户的连接请求，并验证用户口令
2. 接受用户的请求，并将请求信息压入堆栈；
3. 从结果输出队列中搜寻对应的结果信息，并将结果信息发送给客户；
4. 处理需求处理过程中出现的异常，并将日志信息发送给日志服务器。

```

```

*/

import java.io.*;
import java.net.*;

public class ClientThread extends Thread {

    public ClientThread([ParamList]){

    }

    public void synchronized run(){

    }
}

/*
    请求队列：
    1. 将客户的需求压入队列
    2. 将客户的需求弹出队列
*/

import java.util.*;

public class InputQuene {

    private Vector InputTeam;

    public InputQuene() {

        /*初始化队列容量*/
        InputTeam=new Vector(100);

    }

    /*需求进队函数*/
    public synchronized void enQuene(Request request){
        InputTeam.add(request);
    }

    /*将一个请求出队*/
    public synchronized void deQuene(int index){
        this.InputTeam.remove(index);
    }

}

```



```
/*
请求队列处理线程
1. 按先进先出的算法从需求队列中依次取出每一个请求，并进行处理
2. 更新请求的处理状态
3. 清理已经处理过的请求
*/
```

```
import java.io.*;
import java.util.*;
```

```
public class InputProcessThread extends Thread{

    private InputQuene _InQuene;

    public InputProcessThread(){
    }

    public void run(){
    }

}
```

```
/*
结果输出队列：
1. 完成输出结果的进队
2. 完成输出结果的出队
*/
```

```
import java.util.*;
import java.io.*;

public class OutputQuene {

    //结果输出队列容器
    private Vector outputTeam;

    public OutputQuene() {

        //初始化结果输出队列
        outputTeam=new Vector(100);
    }

    //进队函数
    public synchronized void enQuene(Result result){
        outputTeam.add(result);
    }
}
```

```

/*出队函数*/
public synchronized void deQuene(int index){
    outputTeam.remove(index);
}
}

/*
    结果处理线程：
    1。完成输出结果的确认
    2。完成输出结果文件流的生成
    3。完成文件流的压缩处理
*/
import java.io.*;

public class OutputProcessThread extends Thread{

    private OutputQuene _outputQuene;

    public OutputProcessThread([ParamList]) {
        //todo
    }

    /*线程的执行绪*/
    public void run(){
        while(doining){
            try{

                /*处理输出队列*/
                ProcessQuene();

            }catch(Exception e){
                e.printStackTrace();
            }

        }

    }

}

/*
    日志信息处理线程：
    功能说明：

```

1. 完成服务器日志信息的保存
2. 根据设定的规则进行日志信息的清理

期望的目标：

目前日志信息的保存在一个文件当中，以后要自动控制文件的大小。

*/

```
import java.io.*;
```

```
import java.util.*;
```

```
public class LogThread extends Thread{
```

```
    private LogQuene logquene;
```

```
    public LogThread([ParamList]){
```

```
        //todo
```

```
    }
```

```
    /*处理日志信息*/
```

```
    public void run(){
```

```
        while(doin){
```

```
            this.processLog();
```

```
            try{
```

```
                this.sleep(100);
```

```
            }catch(Exception e){
```

```
            }
```

```
        }
```

```
    }
```

```
}
```

```
/* 功能描述：
```

```
    管理缓冲区中的文件信息，将文件所有的大小控制在系统设定的范围之内
```

```
*/
```

```
import java.io.*;
```

```
import java.lang.*;
```

```
import java.util.*;
```

```
import java.text.*;
```

```
import java.math.*;
```

```
public class CacheThread extends Thread{
```

```
    private String CachePath;
```

```
/*类的建构式： 参数：URL 缓冲区目录的路径信息*/
```

```
public CacheThread(String Url) {  
    this.CachePath =Url;  
  
    /*创建文件搜索类*/  
    try{  
        this.CacheDir =new File(this.CachePath);  
    }catch(Exception e){  
        e.printStackTrace();  
    }  
}
```

```
//线程的执行绪
```

```
public void run(){  
    //定时清理缓冲区中的文件  
}
```

```
.....
```

I 如何架构一个“Just so so”的网站？

2007 年 08 月 11 日 星期六 下午 04:13

作者：老王

所谓“Just so so”，翻译成中文大致是“马马虎虎，还算凑合”的意思。所以，如果你想搞一个新浪，搜狐之类的门户的话，估计这篇文章对你没有太大用处，但是就像 80/20 原则所叙述的一样，大多数站点其实都是“Just so so”的规模而已。

那么如何架构一个“Just so so”的网站呢？IMO（在我看来：In My Opinions），可以粗略的分为硬架构和软架构，这个分类是我一拍脑袋杜撰出来的，所以有考

证癖的网友们也不用去搜索引擎查找相关资料了。简单解释一下：所谓硬架构主要是说网站的运行方式和环境等。所谓软架构主要是说在代码层次上如何实现功能等。下面就分别看看 How to do。

一：硬架构

1：机房的选择：

在选择机房的时候，根据网站用户的地域分布，可以选择网通或电信机房，但更多时候，可能双线机房才是合适的。越大的城市，机房价格越贵，从成本的角度看可以在一些中小城市托管服务器，比如说北京的公司可以考虑把服务器托管在天津，廊坊等地，不是特别远，但是价格会便宜很多。

2：带宽的大小：

通常老板花钱请我们架构网站的时候，会给我们提出一些目标，诸如网站每天要能承受 100 万 PV 的访问量等等。这时我们要预算一下大概需要多大的带宽，计算带宽大小主要涉及两个指标（峰值流量和页面大小），我们不妨在计算前先做出必要的假设：

第一：假设峰值流量是平均流量的 5 倍。

第二：假设每次访问平均的页面大小是 100K 字节左右。

如果 100 万 PV 的访问量在一天内平均分布的话，折合到每秒大约 12 次访问，如果按平均每次访问页面的大小是 100K 字节左右计算的话，这 12 次访问总计大约就是 1200K 字节，字节的单位是 Byte，而带宽的单位是 bit，它们之间的关系是 $1\text{Byte} = 8\text{bit}$ ，所以 1200K Byte 大致就相当于 9600K bit，也就是 9Mbps 的样子，实际情况中，我们的网站必须能在峰值流量时保持正常访问，所以按照假设的峰值流量算，真实带宽的需求应该在 45Mbps 左右。

当然，这个结论是建立在前面提到的两点假设的基础上，如果你的实际情况和这两点假设有出入，那么结果也会有差别。

3：服务器的划分：

先看我们都需要哪些服务器：图片服务器，页面服务器，数据库服务器，应用服务器，日志服务器等等。

对于访问量大点的网站而言，分离单独的图片服务器和页面服务器相当必要，我们可以用 lighttpd 来跑图片服务器，用 apache 来跑页面服务器，当然也可以选择别的，甚至，我们可以扩展成很多台图片服务器和很多台页面服务器，并设置相关域名，如 `img.domain.com` 和 `www.domain.com`，页面里的图片路径都使用绝对路径，如 ``，然后设置 DNS 轮循，达到最初级的负载均衡。当然，服务器多了就不可避免的涉及一个同步的问

题，这个可以使用 rsync 软件来搞定。

数据库服务器是重中之重，因为网站的瓶颈问题十有八九是出在数据库身上。现在一般的中小网站多使用 MySQL 数据库，不过它的集群功能似乎还没有达到 stable 的阶段，所以这里不做评价。一般而言，使用 MySQL 数据库的时候，我们应该搞一个主从（一主多从）结构，主数据库服务器使用 innodb 表结构，从数据服务器使用 myisam 表结构，充分发挥它们各自的优势，而且这样的主从结构分离了读写操作，降低了读操作的压力，甚至我们还可以设定一个专门的从服务器做备份服务器，方便备份。不然如果你只有一台主服务器，在大数据量的情况下，mysql dump 基本就没戏了，直接拷贝数据文件的话，还得先停止数据库服务再拷贝，否则备份文件会出错。但对于很多网站而言，即使数据库服务仅停止了一秒也是不可接受的。如果你有了一台从数据库服务器，在备份数据的时候，可以先停止服务（slave stop）再备份，再启动服务（slave start）后从服务器会自动从主服务器同步数据，一切都没有影响。但是主从结构也是有致命缺点的，那就是主从结构只是降低了读操作的压力，却不能降低写操作的压力。为了适应更大的规模，可能只剩下最后这招了：横向/纵向分割数据库。所谓横向分割数据库，就是把不同的表保存到不同的数据库服务器上，比如说用户表保存在 A 数据库服务器上，文章表保存在 B 数据库服务器上，当然这样的分割是有代价的，最基本的就是你没法进行 LEFT JOIN 之类的操作了。所谓纵向分割数据库，一般是指按照用户标识（user_id）等来划分数据存储的服务器，比如说：我们有 5 台数据库服务器，那么“ $\text{user_id} \% 5 + 1$ ”等于 1 的就保存到 1 号服务器，等于 2 的就保存到 2 号服务器，以此类推，纵向分隔的原则有很多种，可以视情况选择。不过和横向分割数据库一样，纵向分割数据库也是有代价的，最基本的就是我们在进行如 COUNT，SUM 等汇总操作的时候会麻烦很多。综上所述，数据库服务器的解决方案一般视情况往往是一个混合的方案，以其发挥各种方案的优势，有时候还需要借助 memcached 之类的第三方软件，以便适应更大访问量的要求。

如果有专门的应用服务器来跑 PHP 脚本是最合适不过的了，那样我们的页面服务器只保存静态页面就可以了，可以给应用服务器设置一些诸如 app.domain.com 之类的域名来和页面服务器加以区别。对于应用服务器，我还是更倾向于使用 prefork 模式的 apache，配上必要的 xcache 之类的 PHP 缓存软件，加载模块要越少越好，除了 mod_rewrite 等必要的模块，不必要的东西统统舍弃，尽量减少 httpd 进程的内存消耗，而那些图片服务器，页面服务器等静态内容就可以使用 lighttpd 或者 tux 来搞，充分发挥各种服务器的特点。

如果条件允许，独立的日志服务器也是必要的，一般小网站的做法都是把页面服务器和日志服务器合二为一了，在凌晨访问量不大的时候 cron 运行前一天的日志计算，不过如果你使用 awstats 之类的日志分析软件，对于百万级访问量而言，即使按天归档，也会消耗很多时间和服务器资源去计算，所以分离单独的日志服务器还是有好处的，这样不会影响正式服务器的工作状态。

二：软架构

1: 框架的选择:

现在的 PHP 框架有很多选择, 比如: CakePHP, Symfony, Zend Framework 等等, 至于应该使用哪一个并没有唯一的答案, 要根据 Team 里团队成员对各个框架的了解程度而定。很多时候, 即使没有使用框架, 一样能写出好的程序来, 比如 Flickr 据说就是用 Pear+Smarty 这样的类库写出来的, 所以, 是否用框架, 用什么框架, 一般不是最重要的, 重要的是我们的编程思想里要有框架的意识。

2: 逻辑的分层:

网站规模到了一定的程度之后, 代码里各种逻辑纠缠在一起, 会给维护和扩展带来巨大的障碍, 这时我们的解决方式其实很简单, 那就是重构, 将逻辑进行分层。通常, 自上而下可以分为表现层, 应用层, 领域层, 持久层。

所谓表现层, 并不仅仅就指模板, 它的范围要更广一些, 所有和表现相关的逻辑都应该被纳入表现层的范畴。比如说某处的字体要显示为红色, 某处的开头要空两格, 这些都属于表现层。很多时候, 我们容易犯的错误就是把本属于表现层的逻辑放到了其他层面去完成, 这里说一个很常见的例子: 我们在列表页显示文章标题的时候, 都会设定一个最大字数, 一旦标题长度超过了这个限制, 就截断, 并在后面显示 “..”, 这就是最典型的表现层逻辑, 但是实际情况, 有很多程序员都是在非表现层代码里完成数据的获取和截断, 然后赋值给表现层模板, 这样的代码最直接的缺点就是同样一段数据, 在这个页面我可能想显示前 10 个字, 再另一个页面我可能想显示前 15 个字, 而一旦我们在程序里固化了这个字数, 也就丧失了可移植性。正确的做法是应该做一个视图助手之类的程序来专门处理此类逻辑, 比如说: Smarty 里的 truncate 就属于这样的视图助手 (不过它那个实现不适合中文)。

所谓应用层, 它的主要作用是定义用户可以做什么, 并把操作结果反馈给表现层。至于如何做, 通常不是它的职责范围 (而是领域层的职责范围), 它会通过委派把如何做的工作交给领域层去处理。在使用 MVC 架构的网站中, 我们可以看到类似下面这样的 URL: domain.com/articles/view/123, 其内部编码实现, 一般就是一个 Articles 控制器类, 里面有一个 view 方法, 这就是一个典型的应用层操作, 因为它定义了用户可以做一个查看的动作。在 MVC 架构中, 有一个准则是这么说的: Rich Model Is Good。言外之意, 就是 Controller 要保持 “瘦” 一些比较好, 进而说明应用层要尽量简单, 不要包括涉及领域内容的逻辑。

所谓领域层, 最直接的解释就是包含领域逻辑的层。它是一个软件的灵魂所在。先来看看什么叫领域逻辑, 简单的说, 具有明确的领域概念的逻辑就是领域逻辑, 比如我们在 ATM 机上取钱, 过程大致是这样的: 插入银联卡, 输入密码, 输入取款金额, 确定, 拿钱, 然后 ATM 吐出一个交易凭条。在这个过程中, 银联卡在 ATM 机器里完成钱从帐户上划拨的过程就是一个领域逻辑, 因为取钱在银行中是一个明确的领域概念, 而 ATM 机吐出一个交易凭条则不是领域逻辑, 而仅是一个应用逻辑, 因为吐出交易凭条并不是银行中一个明确的领域概念, 只是一种技术手段, 对应的, 我们取钱后不吐交易凭条, 而发送一条提醒短信也是可能的, 但并不是一定如此, 如果在实际情况中, 我们要求取款后必须吐出交易凭条, 也就

是说吐出交易凭条已经和取款紧密结合，那么你也可以把吐出交易凭条看作是领域逻辑的一部分，一切都以问题的具体情况而定。在 Eric 那本经典的领域驱动设计中，把领域层分为了五种基本元素：实体，值对象，服务，工厂，仓储。具体可以参阅书中的介绍。领域层最常犯的错误就是把本应属于领域层的逻辑泄露到了其他层次，比如说在一个 CMS 系统，对热门文章的定义是这样的：每天被浏览的次数多于 1000 次，被评论的次数多于 100 次，这样的文章就是热门文章。对于一个 CMS 来说，热门文章这个词无疑是一个重要的领域概念，那么我们如何实现这个逻辑的设计的？你可能会给出类似下面的代码：“SELECT ... FROM ... WHERE 浏览 > 1000 AND 评论 > 100”，没错，这是最简单的实现方式，但是这里需要注意的是“每天被浏览的次数多于 1000 次，被评论的次数多于 100 次”这个重要的领域逻辑被隐藏到了 SQL 语句中，SQL 语句显然不属于领域层的范畴，也就是说，我们的领域逻辑泄露了。

所谓持久层，就是指把我们的领域模型保存到数据库中。因为我们的程序代码是面向对象风格的，而数据库一般是关系型的数据库，所以我们需要把领域模型碾平，才能保存到数据库中，但是在 PHP 里，直到目前还没有非常好的 ORM 出现，所以这方面的解决方案不是特别多，参考 Martin 的企业应用架构模式一书，大致可以使用的方法有行数据入口（Row Data Gateway）或者表数据入口（Table Data Gateway），或者把领域层和持久层合二为一变成活动记录（Active Record）的方式。

I 最便宜的高负载网站架构

关键字：企业应用

1, LVS 做前端四层均衡负载

基于 IP 虚拟分发的规则,不同于 apache,squid 这些 7 层基于 http 协议的反向代理软件, LVS 在性能上往往能得到更好的保证！

2, squid 做前端反向代理加缓存

squid 是业内公认的优秀代理服务器，其缓存能力更让许多高负载网站青睐！（比如新浪，网易等）

使用他，配合 ESI 做 WEB 动态内容及图片缓存，最合适不过了

3, apache 用来处理 php 或静态 html，图片

apache 是业内主流 http 服务器，稳定性与性能都能得到良好保证！

4, JBOSS 用来处理含复杂的业务逻辑的请求

JBOSS 是 red hat 旗下的优秀中间件产品，在 java 开源领域小有名气，并且完全支持 j2ee 规范的，功能非常强大

使用他，既能保证业务流程的规范性，又可以节省开支（免费的）

5, mysql 数据库

使用 mysql 数据库，达到百万级别的数据存储，及快速响应，应该是没问题的

6, memcache 作为分布式缓存
缓存应用数据, 或通过 squid 解析 esi 后, 作为数据载体

LVS

squid + jboss squid + jboss squid + apache

mysql + memcache

最后更新: 2007-02-04 20:36

19:53 | [永久链接](#) | [浏览 \(3574\)](#) | [评论 \(6\)](#) | [收藏](#) | [linux 及网络应用](#) | [进入论坛](#) |

[永久链接](#)

<http://galaxystar.javaeye.com/blog/52178>

评论 共 6 条

[发表评论](#)

[whisper](#) 2007-02-04 20:46

apache 的静态负载能力似乎是靠吃内存换来的
与其 jboss, 还不如 perl 来得方便

[clark](#) 2007-02-05 00:10

可以用 lighttpd 替换 apache
如果只用 servlet 容器, 可以用 resin 替换 jboss
后端配 mysql 群集

[galaxystar](#) 2007-02-05 09:20

为了系统能做到线性可扩展及业务需求的稳定性!
一般考虑用比较成熟的技术!
jboss 本身支持异步消息, 分布事务, AOP, 最近 5.0 的 POJOs 可拔插组件模式比 JMX 更容易维护!
放弃 resin, 用 jboss 也是有道理的!
而 lighttpd 处于起步阶段, 处理 HTTP 静态请求或许是好一点, 但是扩展性, 功能都不是很理想, 没有多年社区支持的 apache 那么强大, N 多的 module 撑着, 用前者太不划算了吧!

[magice](#) 2007-02-05 14:27

jboss 的 EJB 模块基本用不到!

[galaxystar](#) 2007-02-05 20:10

是的, 业务接口, 完全可以用 spring 来代替!
通信也可以抛弃 RMI, 用轻量级的 hessian! 特别是组播, JBOSS 的 JGroup 是 TCP 群发软件中,

比较优秀的！

clark 2007-02-16 11:24

resin 的 servlet 性能比 jboss 的 tomcat 5 要好些。
lighttpd 比 apache 的性能好许多，现在的功能基本满足使用了。
没有特殊需要，可以不用 apache.

I 负载均衡技术全攻略

Internet 的规模每一百天就会增长一倍，客户希望获得 7 天 24 小时的不间断可用性及较快的系统反应时间，而不愿屡次看到某个站点“Server Too Busy”及频繁的系统故障。

网络的各个核心部分随着业务量的提高、访问量和数据流量的快速增长，其处理能力和计算强度也相应增大，使得单一设备根本无法承担。在此情况下，如果扔掉现有设备去做大量的硬件升级，这样将造成现有资源的浪费，而且如果再面临下一次业务量的提升，这又将导致再一次硬件升级的高额成本投入，甚至性能再卓越的设备也不能满足当前业务量的需求。于是，负载均衡机制应运而生。

负载均衡（Load Balance）建立在现有网络结构之上，它提供了一种廉价有效透明的方法扩展网络设备和服务器的带宽、增加吞吐量、加强网络数据处理能力、提高网络的灵活性和可用性。

负载均衡有两方面的含义：首先，大量的并发访问或数据流量分担到多台节点设备上分别处理，减少用户等待响应的时间；其次，单个重负载的运算分担到多台节点设备上做并行处理，每个节点设备处理结束后，将结果汇总，返回给用户，系统处理能力得到大幅度提高。

本文所要介绍的负载均衡技术主要是指在均衡服务器群中所有服务器和应用程序之间流量负载的应用，目前负载均衡技术大多数是用于提高诸如在 Web 服务器、FTP 服务器和其它关键任务服务器上的 Internet 服务器程序的可用性和可伸缩性。

负载均衡技术分类

目前有许多不同的负载均衡技术用以满足不同的应用需求，下面从负载均衡所采用的设备对象、应用的网络层次（指 OSI 参考模型）及应用的地理结构等来分类。

软/硬件负载均衡

软件负载均衡解决方案是指在一台或多台服务器相应的操作系统上安装一个或多个附加软件来实现负载均衡，如 DNS Load Balance, CheckPoint Firewall-1 ConnectControl 等，它的优点是基于特定环境，配置简单，使用灵活，成本低廉，可以满足一般的负载均衡需求。

软件解决方案 缺点也较多，因为每台服务器上安装额外的软件运行会消耗系统不定量的资源，越是功能强大的模块，消耗得越多，所以当连接请求特别大的时候，软件本身会成为 服务器工作成败的一个关键；软件可扩展性并不是很好，受到操作系统的限制；由于操作系统本身的 Bug，往往会引起安全问题。

硬件负载均衡解决方案是直接服务器和外部网络间安装负载均衡设备，这种设备我们通常称之为负载均衡器，由于专门的设备完成专门的任务，独立于操作系统，整体性能得到大量提高，加上多样化的负载均衡策略，智能化的流量管理，可达到最佳的负载均衡需求。

负载均衡器有多种多样的形式，除了作为独立意义上的负载均衡器外，有些负载均衡器集成在交换设备中，置于服务器与 Internet 链接之间，有些则以两块网络适配器将这一功能集成到 PC 中，一块连接到 Internet 上，

一块连接到后端服务器群的内部网络上。

一般而言，硬件负载均衡在功能、性能上优于软件方式，不过成本昂贵。

本地/全局负载均衡

负载均衡从其应用的地理结构上分为本地负载均衡(Local Load Balance)和全局负载均衡(Global Load Balance, 也叫地域负载均衡)，本地负载均衡是指对本地的服务器群做负载均衡，全局负载均衡是指对分别放置在不同的地理位置、有不同网络结构的服务器群间作负载均衡。

本地负载均衡能有效地解决数据流量过大、网络负荷过重的问题，并且不需花费昂贵开支购置性能卓越的服务器，充分利用现有设备，避免服务器单点故障造成数据流量的损失。其有灵活多样的均衡策略把数据流量合理地分配给服务器群内的服务器共同负担。即使是再给现有服务器扩充 升级，也只是简单地增加一个新的服务器到服务群中，而不需改变现有网络结构、停止现有的服务。

全局负载均衡主要用于在一个多区域拥有自己服务器的站点，为了使全球用户只以一个 IP 地址或域名就能访问到离自己最近的服务器，从而获得最快的访问速度，也可用于子公司分散站点分布广的大公司通过 Intranet（企业内部互联网）来达到资源统一合理分配的目的。

全局负载均衡有以下的特点：

实现地理位置无关性，能够远距离为用户提供完全的透明服务。

除了能避免服务器、数据中心等的单点失效，也能避免由于 ISP 专线故障引起的单点失效。

解决网络拥塞问题，提高服务器响应速度，服务就近提供，达到更好的访问质量。

网络层次上的负载均衡

针对网络上负载过重的不同瓶颈所在，从网络的不同层次入手，我们可以采用相应的负载均衡技术来解决现有问题。

随着带宽增加，数据流量不断增大，网络核心部分的数据接口将面临瓶颈问题，原有的单一线路将很难满足需求，而且线路的升级又过于昂贵甚至难以实现，这时就可以考虑采用链路聚合（Trunking）技术。

链路聚合技术（第二层负载均衡）将多条物理链路当作一条单一的聚合逻辑链路使用，网络数据流量由聚合逻辑链路中所有物理链路共同承担，由此在逻辑上增大了链路的容量，使其能满足带宽增加的需求。

现代负载均衡技术通常操作于网络的第四层或第七层。第四层负载均衡将一个 Internet 上合法注册的 IP 地址映射为多个内部服务器的 IP 地址，对每次 TCP 连接请求动态使用其中一个内部 IP 地址，达到负载均衡的目的。在第四层交换机中，此种均衡技术得到广泛的应用，一个目标地址是服务器群 VIP（虚拟 IP, Virtual IP address）连接请求的数据包流经交换机，交换机根据源端和目的 IP 地址、TCP 或 UDP 端口号和一定的负载均衡策略，在服务器 IP 和 VIP 间进行映射，选取服务器群中最好的服务器来处理连接请求。

第七层负载均衡控制应用层服务的内容，提供了一种对访问流量的高层控制方式，适合对 HTTP 服务器群的应用。第七层负载均衡技术通过检查流经的 HTTP 报头，根据报头内的信息来执行负载均衡任务。

第七层负载均衡优点表现在如下几个方面：

通过对 HTTP 报头的检查，可以检测出 HTTP400、500 和 600 系列的错误信息，因而能透明地将连接请求重新定向到另一台服务器，避免应用层故障。

可根据流经的数据类型（如判断数据包是图像文件、压缩文件或多媒体文件格式等），把数据流量引向相应内容的服务器来处理，增加系统性能。

能根据连接请求的类型，如是普通文本、图象等静态文档请求，还是 asp、cgi 等的动态文档请求，把相应的请求引向相应的服务器来处理，提高系统的性能及安全性。

第七层负载均衡受到其所支持的协议限制（一般只有 HTTP），这样就限制了它应用的广泛性，并且检查 HTTP 报头会占用大量的系统资源，势必会影响到系统的性能，在大量连接请求的情况下，负载均衡设备自身容易成为网络整体性能的瓶颈。

负载均衡策略

在实际应用中，我们可能不想仅仅是把客户端的服务请求平均地分配给内部服务器，而不管服务器是否宕机。而是想使 Pentium III 服务器比 Pentium II 能接受更多的服务请求，一台处理服务请求较少的服务器能分配到更多的服务请求，出现故障的服务器将不再接受服务请求直至故障恢复等等。

选择合适的负载均衡策略，使多个设备能很好的共同完成任务，消除或避免现有网络负载分布不均、数据流量拥挤反应时间长的瓶颈。在各负载均衡方式中，针对不同的应用需求，在 OSI 参考模型的第二、三、四、七层的负载均衡都有相应的负载均衡策略。

负载均衡策略的优劣及其实现的难易程度有两个关键因素：一、负载均衡算法，二、对网络系统状况的检测方式和能力。

考虑到服务请求的不同类型、服务器的不同处理能力以及随机选择造成的负载分配不均匀等问题，为了更加合理的把负载分配给内部的多个服务器，就需要应用相应的能够正确反映各个服务器处理能力及网络状态的负载均衡算法：

轮循均衡（Round Robin）：每一次来自网络的请求轮流分配给内部中的服务器，从 1 至 N 然后重新开始。此种均衡算法适合于服务器组中的所有服务器都有相同的软硬件配置并且平均服务请求相对均衡的情况。

权重轮循均衡（Weighted Round Robin）：根据服务器的不同处理能力，给每个服务器分配不同的权值，使其能够接受相应权值数的服务请求。例如：服务器 A 的权值被设计成 1，B 的权值是 3，C 的权值是 6，则服务器 A、B、C 将分别接受到 10%、30%、60% 的服务请求。此种均衡算法能确保高性能的服务器得到更多的使用率，避免低性能的服务器负载过重。

随机均衡（Random）：把来自网络的请求随机分配给内部中的多个服务器。

权重随机均衡（Weighted Random）：此种均衡算法类似于权重轮循算法，不过在处理请求分担时是个随机选择的过程。

响应速度均衡（Response Time）：负载均衡设备对内部各服务器发出一个探测请求（例如 Ping），然后根据内部中各服务器对探测请求的最快响应时间来决定哪一台服务器来响应客户端的服务请求。此种均衡算法能较好的反映服务器的当前运行状态，但这最快响应时间仅仅指的是负载均衡设备与服务器间的最快响应时间，而不是客户端与服务 器间的最快响应时间。

最少连接数均衡（Least Connection）：客户端的每一次请求服务在服务器停留的时间可能会有较大的差异，随着工作时间加长，如果采用简单的轮循或随机均衡算法，每一台服务器上的连接进程可能会产生极大的不同，并没有达到真正的负载均衡。最少连接数均衡算法对内部中需负载的每一台服务器都有一个数据记录，记录当前该服务器正在处理的连接数量，当有新的服务连接请求时，将把当前请求分配给连接数最少的服务器，使均衡更加符合实际情况，负载更加均衡。此种均衡算法适合长时处理的请求服务，如 FTP。

处理能力均衡：此种均衡算法将把服务请求分配给内部中处理负荷（根据服务器 CPU 型号、CPU 数量、内存大小及当前连接数等换算而成）最轻的服务器，由于考虑到了内部服务器的处理能力及当前网络运行状况，所以此种均衡算法相对来说更加精确，尤其适合运用到第七层（应用层）负载均衡的情况下。

DNS 响应均衡（Flash DNS）：在 Internet 上，无论是 HTTP、FTP 或是其它的服务请求，客户端一般都是通过域名解析来找到服务器确切的 IP 地址的。在此均衡算法下，分处在不同地理位置的负载均衡设备收到同一个客户端的域名解析请求，并在同一时间内把此域名解析成各自相对应服务器的 IP 地址（即与此负载均衡设备在同一地理位置的服务器的 IP 地址）并返回给客户端，则客户端将以最先收到的域名解析 IP 地址来继续请求服务，而忽略其它的 IP 地址响应。在种均衡策略适合应用在全局负载均衡的情况下，对本地负载均衡是没有意义的。

尽管有多种的负载均衡算法可以较好的把数据流量分配给服务器去负载，但如果负载均衡策略没有对网络系统状况的检测方式和能力，一旦在某台服务器或某段负载均衡设备与服务器网络间出现故障的情况下，负载均衡设备依然把一部分数据流量引向那台服务器，这势必造成大量的服务请求被丢失，达不到不间断可用性的要求。所以良好的负载均衡策略应有对网络故障、服务器系统故障、应用服务故障的检测方式和能力：

Ping 侦测：通过 ping 的方式检测服务器及网络系统状况，此种方式简单快速，但只能大致检测出网络及服务器上的操作系统是否正常，对服务器上的应用服务检测就无能为力了。

TCP Open 侦测：每个服务都会开放某个通过 TCP 连接，检测服务器上某个 TCP 端口（如 Telnet 的 23 口，HTTP 的 80 口等）是否开放来判断服务是否正常。

HTTP URL 侦测：比如向 HTTP 服务器发出一个对 main.html 文件的访问请求，如果收到错误信息，则认为服务器出现故障。

负载均衡策略的优劣除受上面所讲的两个因素影响外，在有些应用情况下，我们需要将来自同一客户端的所有请求都分配给同一台服务器去负担，例如服务器将客户端注册、购物等服务请求信息保存的本地数据库的情况下，把客户端的子请求分配给同一台服务器来处理就显的至关重要了。有两种方式可以解决此问题，一是根据 IP 地址把来自同一客户端的多次请求分配给同一台服务器处理，客户端 IP 地址与服务器的对应信息是保存在负载均衡设备上的；二是在客户端浏览器 cookie 内做独一无二的标识来把多次请求分配给同一台服务器处理，适合通过代理服务器上网的客户端。

还有一种路径外返回模式（Out of Path Return），当客户端连接请求发送给负载均衡设备的时候，中心负载均衡设备将请求引向某个服务器，服务器的回应请求不再返回给中心负载均衡设备，即绕过流量分配器，直接返回给客户端，因此中心负载均衡设备只负责接受并转发请求，其网络负担就减少了很多，并且给客户端提供了更快的响应时间。此种模式一般用于 HTTP 服务器群，在各服务器上要安装一块虚拟网络适配器，并将其 IP 地址设为服务器群的 VIP，这样才能在服务器直接回应客户端请求时顺利的达成三次握手。

负载均衡实施要素

负载均衡方案应是在网站建设初期就应考虑的问题，不过有时随着访问流量的爆炸性增长，超出决策者的意料，这也就成为不得不面对的问题。当我们在引入某种负载均衡方案乃至具体实施时，像其他的许多方案一样，首先是确定当前及将来的应用需求，然后在代价与收效之间做出权衡。

针对当前及将来的应用需求，分析网络瓶颈的不同所在，我们就需要确立是采用哪一类的负载均衡技术，采用什么样的均衡策略，在可用性、兼容性、安全性等等方面要满足多大的需求，如此等等。

不管负载均衡方案是采用花费较少的软件方式，还是购买代价高昂在性能功能上更强的第四层交换机、负载均衡器等硬件方式来实现，亦或其他种类不同的均衡技术，下面这几项都是我们在引入均衡方案时可能要考虑的问题：

性能：性能是我们在引入均衡方案时需要重点考虑的问题，但也是一个最难把握的问题。衡量性能时可将每秒钟通过网络的数据包数目做为一个参数，另一个参数是均衡方案中服务器群所能处理的最大并发连接数目，但是，假设一个均衡系统能处理百万计的并发连接数，可是却只能以每秒 2 个包的速率转发，这显然是没有任何作用的。性能的优劣与负载均衡设备的处理能力、采用的均衡策略息息相关，并且有两点需要注意：一、均衡方案对服务器群整体的性能，这是响应客户端连接请求速度的关键；二、负载均衡设备自身的性能，避免有大量连接请求时自身性能不足而成为服务瓶颈。有时我们也可以考虑采用混合型负载均衡策略来提升服务器群的总体性能，如 DNS 负载均衡与 NAT 负载均衡相结合。另外，针对有大量静态文档请求的站点，也可以考虑采用高速缓存技术，相对来说更节省费用，更能提高响应性能；对有大量 ssl/xml 内容传输的站点，更应考虑采用 ssl/xml 加速技术。

可扩展性：IT 技术日新月异，一年以前最新的产品，现在或许已是网络中性能最低的产品；业务量的急速上升，一年前的网络，现在需要新一轮的扩展。合适的均衡解决方案应能满足这些需求，能均衡不同操作系统和硬件平台之间的负载，能均衡 HTTP、邮件、新闻、代理、数据库、防火墙和 Cache 等不同服务器的负载，并且能以对客户端完全透明的方式动态增加或删除某些资源。

灵活性：均衡解决方案应能灵活地提供不同的应用需求，满足应用需求的不断变化。在不同的服务器群有不同的应用需求时，应有多样的均衡策略提供更广泛的选择。

可靠性：在对服务质量要求较高的站点，负载均衡解决方案应能为服务器群提供完全的容错性和高可用性。但在负载均衡设备自身出现故障时，应该有良好的冗余解决方案，提高可靠性。使用冗余时，处于同一个冗余单元的多个负载均衡设备必须具有有效的方式以便互相进行监控，保护系统尽可能地避免遭受到重大故障的损失。

易管理性：不管是通过软件还是硬件方式的均衡解决方案，我们都希望它有灵活、直观和安全的 management 方式，这样便于安装、配置、维护和监控，提高工作效率，避免差错。在硬件负载均衡设备上，目前主要有三种管理方式可供选择：一、命令行接口（CLI：Command Line Interface），可通过超级终端连接负载均衡设备串行接口来管理，也能 telnet 远程登录管理，在初始化配置时，往往要用到前者；二、图形用户接口（GUI：Graphical User Interfaces），有基于普通 web 页的管理，也有通过 Java Applet 进行安全管理，一般都需要管理端安装有某个版本的浏览器；三、SNMP（Simple Network Management Protocol，简单网络管理协议）支持，通过第三方网络管理软件对符合 SNMP 标准的设备进行管理。

负载均衡配置实例

DNS 负载均衡

DNS 负载均衡技术是在 DNS 服务器中为同一个主机名配置多个 IP 地址，在应答 DNS 查询时，DNS 服务器对每个查

询将以 DNS 文件中主机记录的 IP 地址按顺序返回不同的解析结果，将客户端的访问引导到不同的机器上去，使得不同的客户端访问不同的服务器，从而达到负载均衡的目的。

DNS 负载均衡的优点是经济简单易行，并且服务器可以位于 internet 上任意的位置。但它也存在不少缺点：

为了使本 DNS 服务器和其他 DNS 服务器及时交互，保证 DNS 数据及时更新，使地址能随机分配，一般都要将 DNS 的刷新时间设置的较小，但太小将会使 DNS 流量大增造成额外的网络问题。

一旦某个服务器出现故障，即使及时修改了 DNS 设置，还是要等待足够的时间（刷新时间）才能发挥作用，在此期间，保存了故障服务器地址的客户计算机将不能正常访问服务器。

DNS 负载均衡采用的是简单的轮循负载算法，不能区分服务器的差异，不能反映服务器的当前运行状态，不能做到为性能较好的服务器多分配请求，甚至会出现客户请求集中在某一台服务器上的情况。

要给每台服务器分配一个 internet 上的 IP 地址，这势必会占用过多的 IP 地址。

判断一个站点是否采用了 DNS 负载均衡的最简单方式就是连续的 ping 这个域名，如果多次解析返回的 IP 地址不相同的话，那么这个站点就很可能采用的就是较为普遍的 DNS 负载均衡。但也不一定，因为如果采用的是 DNS 响应均衡，多次解析返回的 IP 地址也可能会不相同。不妨试试 Ping 一下 www.yesky.com，www.sohu.com，www.yahoo.com

现假设有三台服务器来应对 www.test.com 的请求。在采用 BIND 8.x DNS 服务器的 unix 系统上实现起来比较简单，只需在该域的数据记录中添加类似下面的结果：

```
www1 IN A 192.1.1.1
www2 IN A 192.1.1.2
www3 IN A 192.1.1.3
www IN CNAME www1
www IN CNAME www2
www IN CNAME www3
```

在 NT 下的实现也很简单，下面详细介绍在 win2000 server 下实现 DNS 负载均衡的过程，NT4.0 类似：

打开“管理工具”下的“DNS”，进入 DNS 服务配置控制台。

打开相应 DNS 服务器的“属性”，在“高级”选项卡的“服务器选项”中，选中“启用循环”复选框。此步相当于在注册表记录 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\DNS\Parameters 中添加一个双字节制值（dword 值）RoundRobin，值为 1。

打开正向搜索区域的相应区域（如 test.com），新建主机添加主机（A）资源记录，记录如下：

www IN A 192.1.1.1

www IN A 192.1.1.2

www IN A 192.1.1.3

在这里可以看到的区别是在 NT 下一个主机名对应多个 IP 地址记录，但在 uni x 下，是先添加多个不同的主机名分别对应个自的 IP 地址，然后再把这些主机赋同一个别名（CNAME）来实现的。

在此需要注意的是，NT 下本地子网优先级会取代多宿主名称的循环复用，所以在测试时，如果做测试用的客户机 IP 地址与主机资源记录的 IP 在同一有类掩码范围内，就需要清除在“高级”选项卡“服务器选项”中的“启用 netmask 排序”。

NAT 负载均衡

NAT（Network Address Translation 网络地址转换）简单地说就是将一个 IP 地址转换为另一个 IP 地址，一般用于未经注册的内部地址与合法的、已获注册的 Internet IP 地址间进行转换。适用于解决 Internet IP 地址紧张、不想让网络外部知道内部网络结构等的场合下。每次 NAT 转换势必会增加 NAT 设备的开销，但这种额外的开销对于大多数网络来说都是微不足道的，除非在高带宽有大量 NAT 请求的网络上。

NAT 负载均衡将一个外部 IP 地址映射为多个内部 IP 地址，对每次连接请求动态地转换为一个内部服务器的地址，将外部连接请求引到转换得到地址的那个服务器上，从而达到负载均衡的目的。

NAT 负载均衡是一种比较完善的负载均衡技术，起着 NAT 负载均衡功能的设备一般处于内部服务器到外部网间的网关位置，如路由器、防火墙、四层交换机、专用负载均衡器等，均衡算法也较灵活，如随机选择、最少连接数及响应时间等来分配负载。

NAT 负载均衡可以通过软硬件方式来实现。通过软件方式来实现 NAT 负载均衡的设备往往受到带宽及系统本身处理能力的限制，由于 NAT 比较接近网络的低层，因此就可以将它集成在硬件设备中，通常这样的硬件设备是第四层交换机和专用负载均衡器，第四层交换机的一项重要功能就是 NAT 负载均衡。

下面以实例介绍一下 Cisco 路由器 NAT 负载均衡的配置：

现有一台有一个串行接口和一个 Ethernet 接口的路由器，Ethernet 口连接到内部网络，内部网络上有三台 web 服务器，但都只是低端配置，为了处理好来自 Internet 上大量的 web 连接请求，因此需要在此路由器上做 NAT 负载均衡配置，把发送到 web 服务器合法 Internet IP 地址的报文转换成这三台服务器的内部本地地址。其具体配置过程如下：

做好路由器的基本配置，并定义各个接口在做 NAT 时是内部还是外部接口。

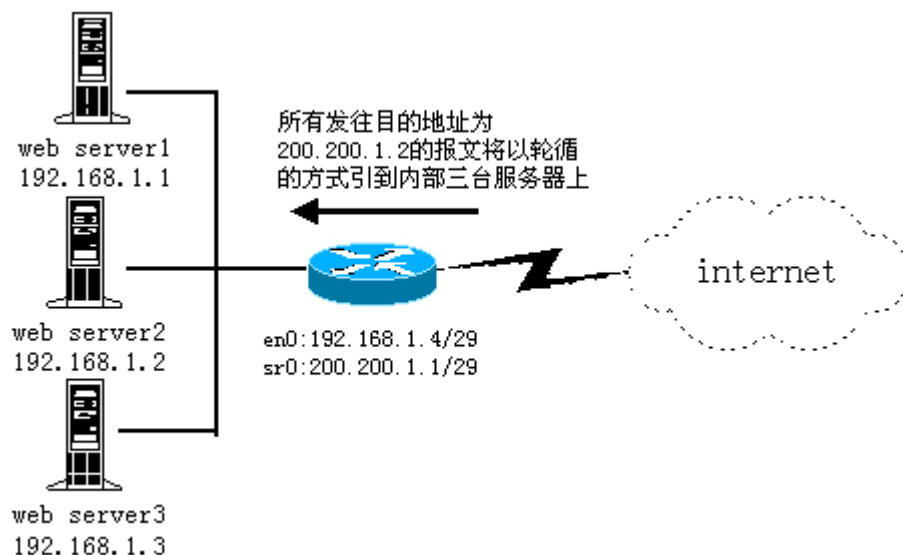
然后定义一个标准访问列表（standard access list），用来标识要转换的合法 IP 地址。

再定义 NAT 地址池来标识内部 web 服务器的本地地址，注意要用到关键字 rotary，表明我们要使用轮循（Round Robin）的方式从 NAT 地址池中取出相应 IP 地址来转换合法 IP 报文。

最后，把目标地址为访问表中 IP 的报文转换成地址池中定义的 IP 地址。

相应配置文件如下：

```
interface Ethernet0/0
ip address 192.168.1.4 255.255.255.248
ip nat inside
!
interface Serial0/0
ip address 200.200.1.1 255.255.255.248
ip nat outside
!
ip access-list 1 permit 200.200.1.2
!
ip nat pool webserv 192.168.1.1 192.168.1.3 netmask 255.255.255.248 type rotary
ip nat inside destination list 1 pool webserv
```



反向代理负载均衡

普通代理方式是代理内部网络用户访问 internet 上服务器的连接请求，客户端必须指定代理服务器，并将本来要直接发送到 internet 上服务器的连接请求发送给代理服务器处理。

反向代理（Reverse Proxy）方式是指以代理服务器来接受 internet 上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给 internet 上请求连接的客户端，此时代理服务器对外就表现为一个服务器。

反向代理负载均衡技术是把将来自 internet 上的连接请求以反向代理的方式动态地转发给内部网络上的多台服务器进行处理，从而达到负载均衡的目的。

反向代理负载均衡能以软件方式来实现，如 apache mod_proxy、netscape proxy 等，也可以在高速缓存器、负

载均衡器等硬件设备上实现。反向代理负载均衡可以将优化的负载均衡策略和代理服务器的高速缓存技术结合在一起，提升静态网页的访问速度，提供有益的性能；由于网络外部用户不能直接访问真实的服务器，具备额外的安全性（同理，NAT 负载均衡技术也有此优点）。

其缺点主要表现在以下两个方面：

反向代理是处于 OSI 参考模型第七层应用的，所以就必须为每一种应用服务专门开发一个反向代理服务器，这样就限制了反向代理负载均衡技术的应用范围，现在一般都用于对 web 服务器的负载均衡。

针对每一次代理，代理服务器就必须打开两个连接，一个对外，一个对内，因此在并发连接请求数量非常大的时候，代理服务器的负载也就非常大了，在最后代理服务器本身会成为服务的瓶颈。

一般来讲，可以用它来对连接数量不是特别大，但每次连接都需要消耗大量处理资源的站点进行负载均衡，如 search。

下面以在 apache mod_proxy 下做的反向代理负载均衡为配置实例：在站点 www.test.com，我们按提供的内容进行分类，不同的服务器用于提供不同的内容服务，将对 <http://www.test.com/news> 的访问转到 IP 地址为 192.168.1.1 的内部服务器上处理，对 <http://www.test.com/it> 的访问转到服务器 192.168.1.2 上，对 <http://www.test.com/life> 的访问转到服务器 192.168.1.3 上，对 <http://www.test.com/love> 的访问转到合作站点 <http://www.love.com> 上，从而减轻本 apache 服务器的负担，达到负载均衡的目的。

首先要确定域名 www.test.com 在 DNS 上的记录对应 apache 服务器接口上具有 internet 合法注册的 IP 地址，这样才能使 internet 上对 www.test.com 的所有连接请求发送给本台 apache 服务器。

在本台服务器的 apache 配置文件 httpd.conf 中添加如下设置：

```
proxypass /news http://192.168.1.1
proxypass /it http://192.168.1.2
proxypass /life http://192.168.1.3
proxypass /love http://www.love.com
```

注意，此项设置最好添加在 httpd.conf 文件“Section 2”以后的位置，服务器 192.168.1.1-3 也应是具有相应功能的 www 服务器，在重启服务时，最好用 apachectl configtest 命令检查一下配置是否有误。

混合型负载均衡

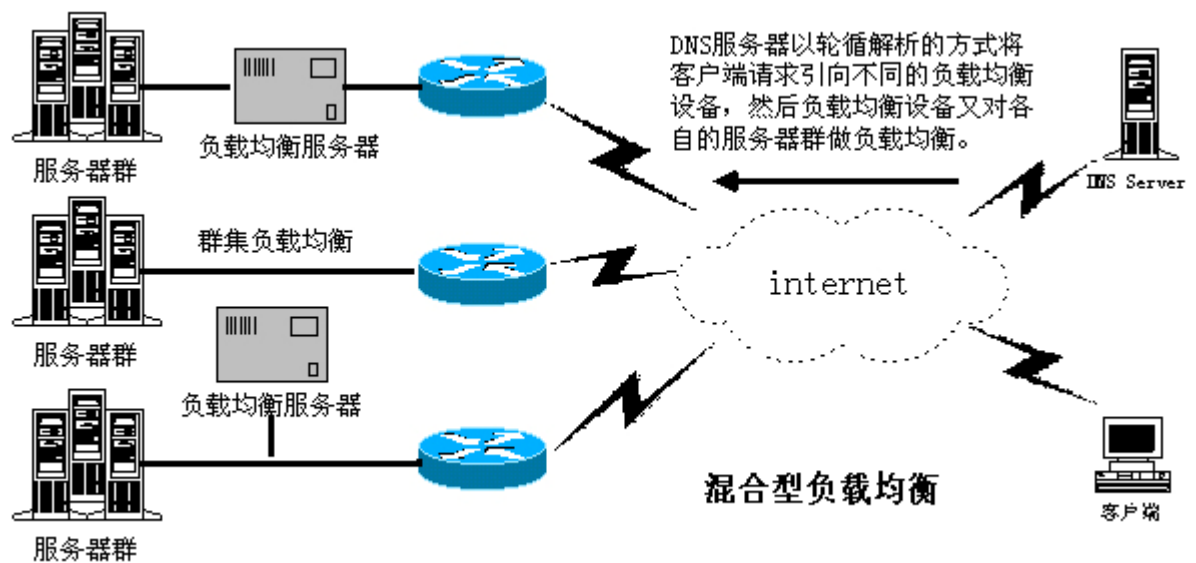
在有些大型网络，由于多个服务器群内硬件设备、各自的规模、提供的服务等差异，我们可以考虑给每个服务器群采用最合适的负载均衡方式，然后又在这多个服务器群间再一次负载均衡或群集起来以一个整体向外界提供服务（即把这多个服务器群当做一个新的服务器群），从而达到最佳的性能。我们将这种方式称之为混合型负载均衡。此种方式有时也用于单台均衡设备的性能不能满足大量连接请求的情况下。

下图展示了一个应用示例，三个服务器群针对各自的特点，分别采用了不同的负载均衡方式。当客户端发出域名解析请求时，DNS 服务器依次把它解析成三个服务器群的 VIP，如此把客户端的连接请求分别引向三个服务器群，从而达到了再一次负载均衡的目的。

在图中大家可能注意到，负载均衡设备在网络拓扑上，可以处于外部网和内部网络间网关的位置，也可以和内部服务器群处于并行的位置，甚至可以处于内部网络或 internet 上的任意位置，特别是在采用群集负载均衡时，根本就没有单独的负载均衡设备。

服务器群内各服务器只有提供相同内容的服务才有负载均衡的意义，特别是在 DNS 负载均衡时。要不然，这样会造成大量连接请求的丢失或由于多次返回内容的不同给客户造成混乱。

所以，如图的这个示例在实际中可能没有多大的意义，因为如此大的服务内容相同但各服务器群存在大量差异的网站并不多见。但做为一个示例，相信还是很有参考意义的。



09:12 | [永久链接](#) | 浏览 (700) | 评论 (3) | 收藏 | [linux 及网络应用](#) |

永久链接

<http://galaxystar.javaeye.com/blog/50542>

评论 共 3 条

[发表评论](#)

[hermitte](#) 2007-02-04 15:09

好文章，收藏下。

原来搞负载平衡，都是用 DNS，对用 NAT 的很感兴趣，仔细读读

[hermitte](#) 2007-02-04 15:14

看完了。。收回评论。。

没什么新的东西。

对于 CGI 程序本身的跨域的状态保持，没有好的办法吗？

[galaxystar](#) 2007-02-04 20:05

跨域，看你的使用情况！

简单的网站，同一域名跨二级域名好办！cookie 里就能支持！
跨一级域名，只能用 SS0 了！比较流行的是 yale 大学的 CAS, 不过性能巨差，我公司里有一套自己搞的！

I 海量数据处理分析

北京迈思奇科技有限公司 戴子良

笔者在实际工作中，有幸接触到海量的数据处理问题，对其进行处理是一项艰巨而复杂的任务。原因有以下几个方面：

一、数据量过大，数据中什么情况都可能存在。如果说有 **10** 条数据，那么大不了每条去逐一检查，人为处理，如果有上百条数据，也可以考虑，如果数据上到千万级别，甚至过亿，那不是手工能解决的了，必须通过工具或者程序进行处理，尤其海量的数据中，什么情况都可能存在，例如，数据中某处格式出了问题，尤其在程序处理时，前面还能正常处理，突然到了某个地方问题出现了，程序终止了。

二、软硬件要求高，系统资源占用率高。对海量的数据进行处理，除了好的方法，最重要的就是合理使用工具，合理分配系统资源。一般情况，如果处理的数据过 **TB** 级，小型机是要考虑的，普通的机器如果有好的方法可以考虑，不过也必须加大 **CPU** 和内存，就象面对着千军万马，光有勇气没有一兵一卒是很难取胜的。

三、要求很高的处理方法和技巧。这也是本文的写作目的所在，好的处理方法是一位工程师长期工作经验的积累，也是个人的经验的总结。没有通用的处理方法，但有通用的原理和规则。

那么处理海量数据有哪些经验和技巧呢，我把我所知道的罗列一下，以供大家参考：

一、选用优秀的数据库工具

现在的数据库工具厂家比较多，对海量数据的处理对所使用的数据库工具要求比较高，一般使用 **Oracle** 或者 **DB2**，微软公司最近发布的 **SQL Server 2005** 性能也不错。另外在 **BI** 领域：数据库，数据仓库，多维数据库，数据挖掘等相关工具也要进行选择，象好的 **ETL** 工具和好的 **OLAP** 工具都十分必要，例如 **Informatic**，**Eassbase** 等。笔者在实际数据分析项目中，对每天 **6000** 万条的日志数据进行处理，使用 **SQL Server 2000** 需要花费 **6** 小时，而使用 **SQL Server 2005** 则只需要花费 **3** 小时。

二、编写优良的程序代码

处理数据离不开优秀的程序代码，尤其在处理复杂数据时，必须使用程序。好的程序代码对数据的处理至关重要，这不仅仅是数据处理准确度的问题，更是数据处理效率的问题。良好的程序代码应该包含好的算法，包含好的处理流程，包含好的效率，包含好的异常处理机制等。

三、对海量数据进行分区操作

对海量数据进行分区操作十分必要，例如针对按年份存取的数据，我们可以按年进行分区，不同的数据库有不同的分区方式，不过处理机制大体相同。例如 **SQL Server** 的数据库分区是

将不同的数据存于不同的文件组下,而不同的文件组存于不同的磁盘分区下,这样将数据分散开,减小磁盘 I/O,减小了系统负荷,而且还可以将日志,索引等放于不同的分区下。

四、建立广泛的索引

对海量的数据处理,对大表建立索引是必行的,建立索引要考虑到具体情况,例如针对大表的分组、排序等字段,都要建立相应索引,一般还可以建立复合索引,对经常插入的表则建立索引时要小心,笔者在处理数据时,曾经在一个 ETL 流程中,当插入表时,首先删除索引,然后插入完毕,建立索引,并实施聚合操作,聚合完成后,再次插入前还是删除索引,所以索引要用到好的时机,索引的填充因子和聚集、非聚集索引都要考虑。

五、建立缓存机制

当数据量增加时,一般的处理工具都要考虑到缓存问题。缓存大小设置的好差也关系到数据处理的成败,例如,笔者在处理 2 亿条数据聚合操作时,缓存设置为 100000 条/Buffer,这对于这个级别的数据量是可行的。

六、加大虚拟内存

如果系统资源有限,内存提示不足,则可以靠增加虚拟内存来解决。笔者在实际项目中曾经遇到针对 18 亿条的数据进行处理,内存为 1GB,1 个 P4 2.4G 的 CPU,对这么大的数据量进行聚合操作是有问题的,提示内存不足,那么采用了加大虚拟内存的方法来解决,在 6 块磁盘分区上分别建立了 6 个 4096M 的磁盘分区,用于虚拟内存,这样虚拟的内存则增加为 $4096 * 6 + 1024 = 25600 \text{ M}$,解决了数据处理中的内存不足问题。

七、分批处理

海量数据处理难因为数据量大,那么解决海量数据处理难的问题其中一个技巧是减少数据量。可以对海量数据分批处理,然后处理后的数据再进行合并操作,这样逐个击破,有利于小数据量的处理,不至于面对大数据量带来的问题,不过这种方法也要因时因势进行,如果不允许拆分数据,还需要另想办法。不过一般的数据按天、按月、按年等存储的,都可以采用先分后合的方法,对数据进行分开处理。

八、使用临时表和中间表

数据量增加时,处理中要考虑提前汇总。这样做的目的是化整为零,大表变小表,分块处理完成后,再利用一定的规则进行合并,处理过程中的临时表的使用和中间结果的保存都非常重要,如果对于超海量的数据,大表处理不了,只能拆分为多个小表。如果处理过程中需要多步汇总操作,可按汇总步骤一步步来,不要一条语句完成,一口气吃掉一个胖子。

九、优化查询 SQL 语句

在对海量数据进行查询处理过程中,查询的 SQL 语句的性能对查询效率的影响是非常大的,编写高效优良的 SQL 脚本和存储过程是数据库工作人员的职责,也是检验数据库工作人员水平的一个标准,在对 SQL 语句的编写过程中,例如减少关联,少用或不用游标,设计好高效的数据库表结构等都十分必要。笔者在工作中试着对 1 亿行的数据使用游标,运行 3 个小时没有出结果,这是一定要改用程序处理了。

十、使用文本格式进行处理

对一般的数据处理可以使用数据库,如果对复杂的数据处理,必须借助程序,那么在程序操作数据库和程序操作文本之间选择,是一定要选程序操作文本的,原因为:程序操作文本速度快;对文本进行处理不容易出错;文本的存储不受限制等。例如一般的海量的网络日志都是文本格式或者 csv 格式(文本格式),对它进行处理牵扯到数据清洗,是要利用程序进行处理的,而不建议导入数据库再做清洗。

十一、定制强大的清洗规则和出错处理机制

海量数据中存在着不一致性，极有可能出现某处的瑕疵。例如，同样的数据中的时间字段，有的可能为非标准的时间，出现的原因可能为应用程序的错误，系统的错误等，这是在进行数据处理时，必须制定强大的数据清洗规则和出错处理机制。

十二、 建立视图或者物化视图

视图中的数据来源于基表，对海量数据的处理，可以将数据按一定的规则分散到各个基表中，查询或处理过程中可以基于视图进行，这样分散了磁盘 I/O，正如 10 根绳子吊着一根柱子和一根吊着一根柱子的区别。

十三、 避免使用 32 位机子（极端情况）

目前的计算机很多都是 32 位的，那么编写的程序对内存的需要便受限制，而很多的海量数据处理是必须大量消耗内存的，这便要求更好性能的机子，其中对位数的限制也十分重要。

十四、 考虑操作系统问题

海量数据处理过程中，除了对数据库，处理程序等要求比较高以外，对操作系统的要求也放到了重要的位置，一般是必须使用服务器的，而且对系统的安全性和稳定性等要求也比较高。尤其对操作系统自身的缓存机制，临时空间的处理等问题都需要综合考虑。

十五、 使用数据仓库和多维数据库存储

数据量加大是一定要考虑 OLAP 的，传统的报表可能 5、6 个小时出来结果，而基于 Cube 的查询可能只需要几分钟，因此处理海量数据的利器是 OLAP 多维分析，即建立数据仓库，建立多维数据集，基于多维数据集进行报表展现和数据挖掘等。

十六、 使用采样数据，进行数据挖掘

基于海量数据的数据挖掘正在逐步兴起，面对着超海量的数据，一般的挖掘软件或算法往往采用数据抽样的方式进行处理，这样的误差不会很高，大大提高了处理效率和处理的成功率。一般采样时要注意数据的完整性和，防止过大的偏差。笔者曾经对 1 亿 2 千万行的表数据进行采样，抽取出 400 万行，经测试软件测试处理的误差为千分之五，客户可以接受。

还有一些方法，需要在不同的情况和场合下运用，例如使用代理键等操作，这样的好处是加快了聚合时间，因为对数值型的聚合比对字符型的聚合快得多。类似的情况需要针对不同的需求进行处理。

海量数据是发展趋势，对数据分析和挖掘也越来越重要，从海量数据中提取有用信息重要而紧迫，这便要求处理要准确，精度要高，而且处理时间要短，得到有价值信息要快，所以，对海量数据的研究很有前途，也很值得进行广泛深入的研究。

I 一个很有意义的 SQL 的优化过程（一个电子化支局中的大数据量的统计 SQL）

Posted on 2007-04-28 10:47 七郎归来 阅读(37) 评论(0) 编辑 收藏 引用 .

```
select count(distinct v_yjhm)
  from (select v_yjhm
        from zjjk_t_yssj_o_his a
       where n_yjzl > 0
        and d_sjrq between to_date('20070301', 'yyyymmdd') and
                          to_date('20070401', 'yyyymmdd')
        and v_yjzldm like '40%')
```

```

        and not exists(select 'a' from INST_TRIG_ZJJK_T_YSSJ_O b
where a.v_yjtm=b.yjbh)
        --and v_yjtm not in (select yjbh from
INST_TRIG_ZJJK_T_YSSJ_O)
union
select v_yjhm
from zjjk_t_yssj_u_his a
where n_yjzl > 0
and d_sjrq between to_date('20070301', 'yyyymmdd') and
to_date('20070401', 'yyyymmdd')
and v_yjzldm like '40%'
and not exists(select 'a' from INST_TRIG_ZJJK_T_YSSJ_U b
where a.v_yjtm=b.yjbh))
        --and v_yjtm not in (select yjbh from
INST_TRIG_ZJJK_T_YSSJ_U))

```

说明：1、zjjk_t_yssj_o_his 、zjjk_t_yssj_u_his 的 d_sjrq 上都有一个索引了

2、zjjk_t_yssj_o_his 、zjjk_t_yssj_u_his 的 v_yjtm 都为 not null 字段

3、INST_TRIG_ZJJK_T_YSSJ_O、INST_TRIG_ZJJK_T_YSSJ_U 的 yjbh 为 PK

优化建议：

1、什么是 **DISTINCT** ？ 就是分组排序后取唯一值 ， 底层行为 分组排序

2、什么是 **UNION** 、**UNION ALL** ？ **UNION** ： 对多个结果集取 **DISTINCT** ， 生成一个不含重复记录的结果集，返回给前端，**UNION ALL** ： 不对结果集进行去重复操作 底层行为： 分组排序

3、什么是 **COUNT(*)** ？ 累加

4、需要有什么样的索引？ **S_sjrq + v_yjzldm** ： 理由： 假如全省的数据量在表中全部数为 1000 万，查询月数据量为 200 万，1000 万中特快占 50%， 则通过 **between** 时间(d_sjrq)+ 种类(v_yjzldm)，可过滤出约 100 万，这是最好的检索方式了。

5、两表都要进行一次 **NOT EXISTS** 运算，如何做最优？ **NOT EXISTS** 是不好做的运算，但是我们可以合并两次的 **NOT EXISTS** 运算。让这费资源的活只干一次。

综合以上，我们可以如下优化这个 **SQL**：

1、内部的 **UNION** 也是去重复，外部的 **DISTINCT** 也是去重复，可左右去掉一个，建议内部的改为 **UNION ALL** ， 这里稍请注意一下，如果 **V_YJHM** 有 **NULL** 的情况，可能会引起 **COUNT** 值不对实际数的情况。

2、建一个 **D_SJRQ+V_YJZLDM** 的复合索引

3、将两个子查询先 **UNION ALL** 联结 ， 另两个用来做 **NOT EXISTS** 的表也

UNION ALL 联结

4、在 3 的基础上再做 NOT EXISTS

5、将 NOT EXISTS 替换为 NOT IN ，同时加提示 HASH_AJ 做半连接 HASH 运算

6、最后为外层的 COUNT(DISTINCT ... 获得结果数

SQL 书写如下：

```
select count(distinct v_yjhm)
  from (select v_yjtm, v_yjhm
        from zjjk_t_yssj_o_his a
       where n_yjzl > 0
          and d_sjrq between to_date('20070301', 'yyyymmdd') and
                to_date('20070401', 'yyyymmdd')
          and v_yjzldm like '40%'
       union all
       select v_yjtm, v_yjhm
        from zjjk_t_yssj_u_his a
       where n_yjzl > 0
          and d_sjrq between to_date('20070301', 'yyyymmdd') and
                to_date('20070401', 'yyyymmdd')
          and v_yjzldm like '40%'
      ) a
 where a.v_yjtm not IN
 (select /*+ HASH_AJ */
      yjbh
    from (select yjbh
          from INST_TRIG_ZJJK_T_YSSJ_O
         union all
         select yjbh from INST_TRIG_ZJJK_T_YSSJ_U))
```

经过上述改造，原来这个 SQL 的执行时间如果为 **2 分钟** 的话，现在应该

20 秒 足够！

I 如何优化大数据量模糊查询（架构，数据库设置，SQL..）

请各位大虾对如下需求提供点意见：

1. 实时查询某当日或指定时间段的所有交易记录。
2. 实时查询一批记录，查询条件不确定，条件几乎包含所有字段，可自由组合）
3. 查询返回数据量可非常大，百万纪录级。

目前系统采用三层结构，中间层是 **cics**，按目前使用的查询方式，系统资源占用大，速度慢，对实时交易会造成影响。

并且速度明显慢于原 **C/S** 结构，如 **C/S** 结构用 2 秒，现在可能要 10 秒。想征询一下是否有好的解决方案，能使三层结构的批量查询快于 **C/S** 结构的查询。

由于客户的环境是中间件和 **DB** 各一台服务器，所以无法作负载均衡。

由于客户在外地，他们提供的信息有限，我无法做出更多的判断。不过本周我将赴外地，作测试，步骤和 **biti** 的类似。

基本思路是首先确认 2 层和 3 层是否做完全相同的查询，然后比较执行时间，判断瓶颈，以决定对中间层还是对 **db** 和 **sql** 进行优化。

针对你的 3 个条件做一下回答：

1. 可以考虑使用以时间做条件的 **partition**
2. 总有一两个条件选择性高，使用频率又高的，考虑加索引。
3. 既然是 3 层结构，那就不应该把那么高的数据料一次返回给 **Client**，可以考虑把处理过程放在中间层，或者使用分页技术，根据需要分段返回。

I 求助:海量数据处理方法

大家好,我们现在有一个技术问题,不知道能不能帮忙解决?

- 1.我们网站的信息系统,每天新增 100W 条用户数据,不知道如果解决才能查询更新更快,更合理.
- 2.有一个条数据,同时有 1W 个用户查看(并发用户),我们的统计是每次+1,现在数据库更新时有问题了,排队更新,速度太慢.

注:我们用 **Asp.Net (C#)** ,**Sql Server2005** 平台

re: 求助:海量数据处理方法 回复 更多评论

- 1,记住要按需所取,就是用户一次看多少就显示多少,也就是从数据库中取出这些数量的数据,善用你的索引
 - 2,写存储过程,缓存....
 - 3,静态化页面.
 - 5,修改你的逻辑
- 建立数据中心，对数据进行按某个条件建分区索引

I 海量数据库查询方略

老朋友 **Bob** 遇到难题：

"有这样一个系统，每个月系统自动生成一张数据表，表名按业务代码和年月来命名，每张表的数据一个月平均在 8 k 万这样的数据量，但是查询的时候希望能够查到最近三个月的数据，也就

是要从三个数据量非常庞大的表中来把查询的数据汇聚到一起,有什么比较好的办法有比较高的效率?”

这当然是个海量数据库,他还具体的举例子:

“我现在测试的结果是查询慢,数据量大的时候,在查询分析器做查询都比较慢,比如用户输入一个主叫号码,他希望获取最近 3 个月的数据信息,但是后台要根据该主叫号码到最近 3 个月的表中去查数据再汇聚到一起,返回给客户端。”

海量数据对服务器的 CPU, IO 吞吐都是严峻的考验,我的解决之道:

1.从设计初试就考虑拆分数数据库,让数据库变“小”,比如,把将用户按地域划分,或者按 VIP 等级划分。**Bob** 说按地域划分很困难,因为不知道用户的地域;VIP 级别也不知道。其实,主叫号码就带有信息,比如按区号,按手机号码段,甚至就按主叫号码的前两个数字来拆分。数据库小了自然就快了。

2.上面拆分的方法是把数据库变“小”,更强有力的手段是采用分布式计算,举例如下:

1).用三台服务器安装三套相同的数据库系统,数据完全一样;

2).用三个线程同时向三个服务器发起请求,每个服务器各查一个月,然后将数据汇总起来,这样速度提高了 3 倍。

3.索引优化,**Bob** 自己也谈到,可以根据查询创建有效的复合索引,不过索引复杂了,插入数据会变慢,要仔细权衡。我觉得还有个办法:

主叫号码通常是字符串型的,建议改为长整型,这样索引后检索会十分快,因为整型的比较要远远快过字符串的比较。

希望我这个纸上谈兵对你有所帮助。

I SQL Server 2005 对海量数据处理

超大型数据库的大小常常达到数百 GB,有时甚至要用 TB 来计算。而单表的数据量往往会达到上亿的记录,并且记录数会随着时间而增长。这不但影响着数据库的运行效率,也增大数据库的维护难度。除了表的数据量外,对表不同的访问模式也可能会影响性能和可用性。这些问题都可以通过对大表进行合理分区得到很大的改善。当表和索引变得非常大时,分区可以将数据分为更小、更容易管理的部分来提高系统的运行效率。如果系统有多个 CPU 或是多个磁盘子系统,可以通过并行操作获得更好的性能。所以对大表进行分区是处理海量数据的一种十分高效的方法。本文通过一个具体实例,介绍如何创建和修改分区表,以及如何查看分区表。

1 SQL Server 2005

SQL Server 2005 是微软在推出 SQL Server 2000 后时隔五年推出的一个数据库平台,它的数据库引擎为关系型数据和结构化数据提供了更安全可靠存储功能,使用户可以构建和管理用于业务的高可用和高性能的数据应用程序。此外 SQL Server 2005 结合了分析、报表、集成和通知功能。这使企业可以构建和部署经济有效的 BI 解决方案,帮助团队通过记分卡、Dashboard、Web Services 和移动设备将数据应用推向业务的各个领域。无论是开发人员、数据库管理员、信息工作者还是决策者,SQL Server 2005 都可以提供出创新的解决方案,并可从数据中获得更多的益处。

它所带来的新特性,如 T-SQL 的增强、数据分区、服务代理和与 .Net Framework 的集成等,在易管理性、可用性、可伸缩性和安全性等方面都有很大的增强。

2 表分区的具体实现方法

表分区分为水平分区和垂直分区。水平分区将表分为多个表。每个表包含的列数相同,但是行更少。例如,可以将一个包含十亿行的表水平分区成 12 个表,每个小表表示特定年份内一个月的数据。任何需要特定月份数据的查询只需引用相应月份的表。而垂直分区则是将原始表分成多个只包含较少列的表。水平分区是最常用分区方式,本文以水平分区来介绍具体实现方法。

水平分区常用的方法是根据时期和使用对数据进行水平分区。例如本文例子,一个短信发送记录表包含最近一年的数据,但是只定期访问本季度的数据。在这种情况下,可考虑将数据分成四个区,每个区只包含一个季度的数据。

2.1 创建文件组

建立分区表先要创建文件组,而创建多个文件组主要是为了获得好的 I/O 平衡。一般情况下,文件组数最好与分区数相同,并且这些文件组通常位于不同的磁盘上。每个文件组可以由一个或多个文件构成,而每个分区必须映射到一个文件组。一个文件组可以由多个分区使用。为了更好地管理数据(例如,为了获得更精确的备份控制),对分区表应进行设计,以便只有相关数据或逻辑分组的数据位于同一个文件组中。使用 **ALTER DATABASE**,添加逻辑文件组名:

```
ALTER DATABASE [DeanDB] ADD FILEGROUP [FG1]
```

DeanDB 为数据库名称,FG1 文件组名。创建文件组后,再使用 **ALTER DATABASE** 将文件添加到该文件组中:

```
ALTER DATABASE [DeanDB] ADD FILE ( NAME = N'FG1', FILENAME = N'C:\DeanData\FG1.ndf' , SIZE = 3072KB , FILEGROWTH = 1024KB ) TO FILEGROUP [FG1]
```

类似的建立四个文件和文件组,并把每一个存储数据的文件放在不同的磁盘驱动器里。

2.2 创建分区函数

创建分区表必须先确定分区的功能机制,表进行分区的标准是通过分区函数来决定的。创建数据分区函数有 **RANGE** “**LEFT** | **RIGHT**”两种选择。代表每个边界值在局部的哪一边。例如存在四个分区,则定义三个边界点值,并指定每个值是第一个分区的上边界 (**LEFT**) 还是第二个分区的下边界 (**RIGHT**)[1]。代码如下:

```
CREATE PARTITION FUNCTION [SendSMS PF](datetime) AS RANGE RIGHT FOR VALUES ('20070401', '20070701', '20071001')
```

2.3 创建分区方案

创建分区函数后,必须将其与分区方案相关联,以便将分区指向至特定的文件组。就是定义实际存放数据的媒体与各数据块的对应关系。多个数据表可以共用相同的数据分区函数,一般不共用相同的数据分区方案。可以通过不同的分区方案,使用相同的分区函数,使不同的数据表有相同的分区条件,但存放在不同的媒介上。创建分区方案的代码如下:

```
CREATE PARTITION SCHEME [SendSMSPPS] AS PARTITION [SendSMSPPF] TO ([FG1], [FG2], [FG3], [FG4])
```

2.4 创建分区表

建立好分区函数和分区方案后,就可以创建分区表了。分区表是通过定义分区键值和分区方案相联系的。插入记录时,SQL SERVER 会根据分区键值的不同,通过分区函数的定义将数据放到相应的分区。从而把分区函数、分区方案和分区表三者有机的结合起来。创建分区表的代码如下:

```
CREATE TABLE SendSMSLog  
  
([ID] [int] IDENTITY(1,1) NOT NULL,  
  
[IDNum] [nvarchar](50) NULL,  
  
[SendContent] [text] NULL  
  
[SendDate] [datetime] NOT NULL,  
  
) ON SendSMSPPS(SendDate)
```

2.5 查看分区表信息

系统运行一段时间或者把以前的数据导入分区表后,我们需要查看数据的具体存储情况,即每个分区存取的记录数,那些记录存取在那个分区等。我们可以通过\$partition.SendSMSPPF 来查看,代码如下:

```
SELECT $partition.SendSMSPPF(o.SendDate)  
  
AS [Partition Number]  
  
, min(o.SendDate) AS [Min SendDate]  
  
, max(o.SendDate) AS [Max SendDate]  
  
, count(*) AS [Rows In Partition]
```

```
FROM dbo.SendSMSLog AS o

GROUP BY $partition.SendSMSPF(o.SendDate)

ORDER BY [Partition Number]
```

在查询分析器里执行以上脚本,结果如图 1 所示:

图 1 分区表信息

2.6 维护分区

分区的维护主要设计分区的添加、减少、合并和在分区间转换。可以通过 ALTER PARTITION FUNCTION 的选项 SPLIT,MERGE 和 ALTER TABLE 的选项 SWITCH 来实现。SPLIT 会多增加一个分区,而 MEGRE 会合并或者减少分区,SWITCH 则是逻辑地在组间转换分区。

3 性能对比

我们对 2650 万数据,存储空间占用约 4G 的单表进行性能对比,测试环境为 IBM365,CPU 至强 2.7G*2、内存 16G、硬盘 136G*2,系统平台为 Windows 2003 SP1+SQL Server 2005 SP1。测试结果如表 1:

表 1:分区和未分区性能对比表(单位:毫秒)

测试项目	分区	未分区
------	----	-----

1	16546	61466
---	-------	-------

2	13	33
---	----	----

3	20140	61546
---	-------	-------

4	17140	61000
---	-------	-------

说明:

1:根据时间检索某一天记录所耗时间

2:单条记录插入所耗时间

3:根据时间删除某一天记录所耗时间

4:统计每月的记录数所需时间

从表 1 可以看出,对分区表进行操作比未分区的表要快,这是因为对分区表的操作采用了 CPU 和 I/O 的并行操作,检索数据的数据量也变小了,定位数据所耗时间变短。

4 结束语

对海量数据的处理一直是一个令人头痛的问题。分离的技术是所有设计者们首先考虑的问题,不管是分离应用程序功能还是分离数据访问,如果加以了合理规划,都能十分有效的解决大数据表的运行效率低和维护成本高等问题。**SQL Server 2005** 新增的表分区功能,可以对数据进行合理分区,当用户在访问部分数据时,SQL Server 最佳化引擎可以根据数据的实体存放,找出最佳的执行方案,而不至于大海捞针。

I 分表处理设计思想和实现

作者: heiyeluren (黑夜路人)

博客: <http://blog.csdn.net/heiyeshuwu>

时间: 2007-01-19 01:44:20

一、概述

分表是个目前算是比较炒的比较流行的概念,特别是在大负载的情况下,分表是一个良好分散数据库压力的好方法。

首先要了解为什么要分表,分表的好处是什么。我们先来大概了解以下一个数据库执行 SQL 的过程:

接收到 SQL --> 放入 SQL 执行队列 --> 使用分析器分解 SQL --> 按照分析结果进行数据的提取或者修改 --> 返回处理结果

当然,这个流程图不一定正确,这只是我自己主观意识上这么我认为。那么这个处理过程当中,最容易出现问题的是什么?就是说,如果前一个 SQL 没有执行完毕的话,后面的 SQL 是不会执行的,因为为了保证数据的完整性,必须对数据表文件进行锁定,包括共享锁和独享锁两种锁定。共享锁是在锁定的期间,其它线程也可以访问这个数据文件,但是不允许修改操作,相应的,独享锁就是整个文件就是归一个线程所有,其它线程无法访问这个数据文件。一般 MySQL 中最快的存储引擎 MyISAM,它是基于表锁定的,就是说如果一锁定的话,那么整个数据文件外部都无法访问,必须等前一个操作完成后,才能接收下一个操作,那么在这个前一个操作没有执行完成,后一个操作等待在队列里无法执行的情况叫做阻塞,一般我们通俗意义上叫做“锁表”。

锁表直接导致的后果是什么?就是大量的 SQL 无法立即执行,必须等队列前面的 SQL 全部

执行完毕才能继续执行。这个无法执行的 SQL 就会导致没有结果，或者延迟严重，影响用户体验。

特别是对于一些使用比较频繁的表，比如 SNS 系统中的用户信息表、论坛系统中的帖子表等等，都是访问量大很大的表，为了保证数据的快速提取返回给用户，必须使用一些处理方式来解决这个问题，这个就是我今天要聊到的分表技术。

分表技术顾名思义，就是把若干个存储相同类型数据的表分成几个表分表存储，在提取数据的时候，不同的用户访问不同的表，互不冲突，减少锁表的几率。比如，目前保存用户分表有两个表，一个是 user_1 表，还有一个是 user_2 表，两个表保存了不同的用户信息，user_1 保存了前 10 万的用户信息，user_2 保存了后 10 万名用户的信息，现在如果同时查询用户 heiyeluren1 和 heiyeluren2 这个两个用户，那么就是分表从不同的表提取出来，减少锁表的可能。

我下面要讲述的两种分表方法我自己都没有实验过，不保证准确能用，只是提供一个设计思路。下面关于分表的例子我假设是在一个贴吧系统的基础上来进行处理和构建的。（如果没有用过贴吧的用户赶紧 Google 一下）

二、基于基础表的分表处理

这个基于基础表的分表处理方式大致的思想就是：一个主要表，保存了所有的基本信息，如果某个项目需要找到它所存储的表，那么必须从这个基础表中查找出对应的表名等项目，好直接访问这个表。如果觉得这个基础表速度不够快，可以完全把整个基础表保存在缓存或者内存中，方便有效的查询。

我们基于贴吧的情况，构建假设如下的 3 张表：

1. 贴吧版块表：保存贴吧中版块的信息
2. 贴吧主题表：保存贴吧中版块中的主题信息，用于浏览
3. 贴吧回复表：保存主题的原始内容和回复内容

“贴吧版块表”包含如下字段：

版块 ID	board_id	int(10)
版块名称	board_name	char(50)
子表 ID	table_id	smallint(5)
产生时间	created	datetime

“贴吧主题表”包含如下字段：

主题 ID	topic_id	int(10)
-------	----------	---------

主题名称	topic_name	char(255)
版块 ID	board_id	int(10)
创建时间	created	datetime

“贴吧回复表”的字段如下：

回复 ID	reply_id	int(10)
回复内容	reply_text	text
主题 ID	topic_id	int(10)
版块 ID	board_id	int(10)
创建时间	created	datetime

那么上面保存了我们整个贴吧中的表结构信息，三个表对应的关系是：

版块 --> 多个主题
主题 --> 多个回复

那么就是说，表文件大小的关系是：

版块表文件 < 主题表文件 < 回复表文件

所以基本可以确定需要对主题表和回复表进行分表，已增加我们数据检索查询更改时候的速度和性能。

看了上面的表结构，会明显发现，在“版块表”中保存了一个"table_id"字段，这个字段就是用于保存一个版块对应的主题和回复都是分表保存在什么表里的。

比如我们有一个叫做“PHP”的贴吧，board_id 是 1，子表 ID 也是 1，那么这条记录就是：

board_id	board_name	table_id	created
1	PHP	1	2007-01-19 00:30:12

相应的，如果我需要提取“PHP”吧里的所有主题，那么就必须按照表里保存的 table_id 来组合一个存储了主题的表名称，比如我们主题表的前缀是“topic_”，那么组合出来“PHP”吧对应的主题表应该是：“topic_1”，那么我们执行：

```
SELECT * FROM topic_1 WHERE board_id = 1 ORDER BY topic_id DESC LIMIT 10
```

这样就能够获取这个主题下面回复列表，方便我们进行查看，如果需要查看某个主题下面的回复，我们可以继续使用版块表中保存的“table_id”来进行查询。比如我们回复表的前缀是“reply_”，那么就可以组合出“PHP”吧的 ID 为 1 的主题的回复：

```
SELECT * FROM reply_1 WHERE topic_id = 1 ORDER BY reply_id DESC LIMIT 10
```

这里，我们能够清晰的看到，其实我们这里使用了基础表，基础表就是我们的版块表。那么相应的，肯定会说：基础表的数据量大了以后如何保证它的速度和效率？

当然，我们就必须使得这个基础表保持最好的速度和性能，比如，可以采用 MySQL 的内存表来存储，或者保存在内存当中，比如 Memcache 之类的内存缓存等等，可以按照实际情况来进行调整。

一般基于基础表的分表机制在 SNS、交友、论坛等 Web2.0 网站中是个比较不错的解决方案，在这些网站中，完全可以单独使用一个表来保存基本标识和目标表之间的关系。使用表保存对应关系的好处是以后扩展非常方便，只需要增加一个表记录。

【优势】增加删除节点非常方便，为后期升级维护带来很大便利

【劣势】需要增加表或者对某一个表进行操作，还是无法离开数据库，会产生瓶颈

三、基于 Hash 算法的分表处理

我们知道 Hash 表就是通过某个特殊的 Hash 算法计算出的一个值，这个值必须是惟一的，并且能够使用这个计算出来的值查找到需要的值，这个叫做哈希表。

我们在分表里的 hash 算法跟这个思想类似：通过一个原始目标的 ID 或者名称通过一定的 hash 算法计算出数据存储表的表名，然后访问相应的表。

继续拿上面的贴吧来说，每个贴吧有版块名称和版块 ID，那么这两项值是固定的，并且是惟一的，那么我们就可以考虑通过对这两项值中的一项进行一些运算得出一个目标表的名称。

现在假如我们针对我们这个贴吧系统，假设系统最大允许 1 亿条数据，考虑每个表保存 100 万条记录，那么整个系统就不超过 100 个表就能够容纳。按照这个标准，我们假设在贴吧的版块 ID 上进行 hash，获得一个 key 值，这个值就是我们的表名，然后访问相应的表。

我们构造一个简单的 hash 算法：

```
function get_hash($id){
    $str = bin2hex($id);
    $hash = substr($str, 0, 4);
    if (strlen($hash)<4){
        $hash = str_pad($hash, 4, "0");
    }
    return $hash;
}
```

算法大致就是传入一个版块 ID 值，然后函数返回一个 4 位的字符串，如果字符串长度不够，使用 0 进行补全。

比如：get_hash(1)，输出的结果是“3100”，输入：get_hash(23819)，得到的结果是：3233，那么我们经过简单的跟表前缀组合，就能够访问这个表了。那么我们需要访问 ID 为 1 的内容时候哦，组合的表将是：topic_3100、reply_3100，那么就可以直接对目标表进行访问了。

当然，使用 hash 算法后，有部分数据是可能在同一个表的，这一点跟 hash 表不同，hash 表是尽量解决冲突，我们这里不需要，当然同样需要预测和分析表数据可能保存的表名。

如果需要存储的数据更多，同样的，可以对版块的名字进行 hash 操作，比如也是上面的二进制转换成十六进制，因为汉字比数字和字母要多很多，那么重复几率更小，但是可能组合成的表就更多了，相应就必须考虑一些其它的问题。

归根结底，使用 hash 方式的话必须选择一个好的 hash 算法，才能生成更多的表，然数据查询的更迅速。

【优点 hash 算法直接得出目标表名称，效率很高】通过

【劣势】扩展性比较差，选择了一个 hash 算法，定义了多少数据量，以后只能在这个数据量上跑，不能超过这个数据量，可扩展性稍差

四、其它问题

1. 搜索问题

现在我们已经进行分表了，那么就无法直接对表进行搜索，因为你无法对可能系统中已经存在的几十或者几百个表进行检索，所以搜索必须借助第三方的组件来进行，比如 Lucene 作为站内搜索引擎是个不错的选择。

2. 表文件问题

我们知道 MySQL 的 MyISAM 引擎每个表都会生成三个文件，*.frm、*.MYD、*.MYI 三个文件，分表用来保存表结构、表数据和表索引。Linux 下面每个目录下的文件数量最好不要超过 1000 个，不然检索数据将更慢，那么每个表都会生成三个文件，相应的如果分表超过 300 个表，那么将检索非常慢，所以这时候就必须再进行分，比如在进行数据库的分离。

使用基础表，我们可以新增加一个字段，用来保存这个表保存在什么数据。使用 Hash 的方式，我们必须截取 hash 值中第几位来作为数据库的名字。这样，完好的解决这个问题。

五、总结

在大负载应用当中，数据库一直是个很重要的瓶颈，必须要突破，本文讲解了两种分表的方式，希望对很多人能够有启发的作用。当然，本文代码和设想没有经过任何代码测试，所以

无法保证设计的完全准确实用，具体还是需要读者在使用过程当中认真分析实施。

I Linux 系统高负载 MySQL 数据库彻底优化(1)

作者: skid 出处:赛迪网 () 砖 () 好 评论 () 条 进入论坛

更新时间: 2007-06-25 13:43

关 键 词: 优化 Linux MySQL

阅读提示: 本文作者讲述了在高负载的 Linux 系统下, MySQL 数据库如何实现优化, 供大家参考!

同时在线访问量继续增大对于 1G 内存的服务器明显感觉到吃力, 严重时, 甚至每天都会死机或者时不时的服务器卡一下。这个问题曾经困扰了我半个多月, MySQL 使用是很具伸缩性的算法, 因此你通常能用很少的内存运行或给 MySQL 更多的备存以得到更好的性能。

安装好 mysql 后, 配制文件应该在 /usr/local/mysql/share/mysql 目录中, 配制文件有几个, 有 my-huge.cnf my-medium.cnf my-large.cnf my-small.cnf, 不同流量的网站和不同配制的服务器环境, 当然需要有不同的配制文件了。

一般的情况下, my-medium.cnf 这个配制文件就能满足我们的大多需要; 一般我们会把配置文件拷贝到 /etc/my.cnf, 只需要修改这个配置文件就可以了, 使用 `mysqladmin variables extended-status -uroot -p` 可以看到目前的参数, 有 3 个配置参数是最重要的, 即 `key_buffer_size`, `query_cache_size`, `table_cache`。

`key_buffer_size` 只对 MyISAM 表起作用, `key_buffer_size` 指定索引缓冲区的大小, 它决定索引处理的速度, 尤其是索引读的速度。一般我们设为 16M, 实际上稍微大一点的站点 这个数字是远远不够的, 通过检查状态值 `Key_read_requests` 和 `Key_reads`, 可以知道 `key_buffer_size` 设置是否合理。比例 `key_reads / key_read_requests` 应该尽可能的低, 至少是 1:100, 1:1000 更好(上述状态值可以使用 `SHOW STATUS LIKE 'key_read%'` 获得)。或者如果你装了 `phpmyadmin` 可以通过服务器运行状态看到, 笔者推荐用 `phpmyadmin` 管理 mysql, 以下的状态值都是本人通过 `phpmyadmin` 获得的实例分析:

这个服务器已经运行了 20 天

`key_buffer_size` - 128M

`key_read_requests` - 650759289

`key_reads` - 79112

比例接近 1:8000 健康状况非常好

另外一个估计 `key_buffer_size` 的办法

把你网站数据库的每个表的索引所占空间大小加起来看看以此服务器为例: 比较大的几个表索引加起来大概 125M 这个数字会随着表变大而变大。

从 4.0.1 开始, MySQL 提供了查询缓冲机制。使用查询缓冲, MySQL 将 SELECT 语句和查询结果存放在缓冲区中, 今后对于同样的 SELECT 语句(区分大小写), 将直接从缓冲区中读取结果。根据 MySQL 用户手册, 使用查询缓冲最多可以达到 238% 的效率。

通过调节以下几个参数可以知道 query_cache_size 设置得是否合理

Qcache inserts

Qcache hits

Qcache lowmem prunes

Qcache free blocks

Qcache total blocks

Qcache_lowmem_prunes 的值非常大，则表明经常出现缓冲不够的情况，同时 Qcache_hits 的值非常大，则表明查询缓冲使用非常频繁，此时需要增加缓冲大小 Qcache_hits 的值不大，则表明你的查询重复率很低，这种情况下使用查询缓冲反而会影响效率，那么可以考虑不用查询缓冲。此外，在 SELECT 语句中加入 SQL_NO_CACHE 可以明确表示不使用查询缓冲。

Qcache_free_blocks，如果该值非常大，则表明缓冲区中碎片很多，query_cache_type 指定是否使用查询缓冲。

我设置：

QUOTE：

query_cache_size = 32M

query_cache_type= 1

得到如下状态值：

Qcache queries in cache 12737 表明目前缓存的条数

Qcache inserts 20649006

Qcache hits 79060095 看来重复查询率还挺高的

Qcache lowmem prunes 617913 有这么多次出现缓存过低的情况

Qcache not cached 189896

Qcache free memory 18573912 目前剩余缓存空间

Qcache free blocks 5328 这个数字似乎有点大 碎片不少

Qcache total blocks 30953

如果内存允许 32M 应该要往上加点

`table_cache` 指定表高速缓存的大小。每当 MySQL 访问一个表时，如果在表缓冲区中还有空间，该表就被打开并放入其中，这样可以更快地访问表内容。通过检查峰值时间的状态值 `Open_tables` 和 `Opened_tables`，可以决定是否需要增加 `table_cache` 的值。如果你发现 `open_tables` 等于 `table_cache`，并且 `opened_tables` 在不断增长，那么你就需要增加 `table_cache` 的值了（上述状态值可以使用 `SHOW STATUS LIKE 'Open%tables'` 获得）。注意，不能盲目地把 `table_cache` 设置成很大的值。如果设置得太高，可能会造成文件描述符不足，从而造成性能不稳定或者连接失败。

对于有 1G 内存的机器，推荐值是 128—256。

笔者设置

QUOTE:

```
table_cache = 256
```

得到以下状态:

```
Open tables 256
```

```
Opened tables 9046
```

虽然 `open_tables` 已经等于 `table_cache`，但是相对于服务器运行时间来说，已经运行了 20 天，`opened_tables` 的值也非常低。因此，增加 `table_cache` 的值应该用处不大。如果运行了 6 个小时就出现上述值那就要考虑增大 `table_cache`。

如果你不需要记录 2 进制 `log` 就把这个功能关掉，注意关掉以后就不能恢复出问题前的数据了，需要您手动备份，二进制日志包含所有更新数据的语句，其目的是在恢复数据库时，用它来把数据尽可能恢复到最后的状态。另外，如果做同步复制(`Replication`)的话，也需要使用二进制日志传送修改情况。

`log_bin` 指定日志文件，如果不提供文件名，MySQL 将自己产生缺省文件名。MySQL 会在文件名后面自动添加数字引，每次启动服务时，都会重新生成一个新的二进制文件。此外，使用 `log-bin-index` 可以指定索引文件；使用 `binlog-do-db` 可以指定记录的数据库；使用 `binlog-ignore-db` 可以指定不记录的数据库。注意的是：`binlog-do-db` 和 `binlog-ignore-db` 一次只指定一个数据库，指定多个数据库需要多个语句。而且，MySQL 会将所有的数据库名称改成小写，在指定数据库时必须全部使用小写字母，否则不会起作用。

关掉这个功能只需要在他前面加上 # 号

QUOTE:

```
#log-bin
```

数据库的瓶颈大多在查询速度和读写锁定上，除了优化数据库本身和 `sql` 语句外，还可以考虑，把一个表拆分成多个或者关系数据库和文本（纯文本 / XML / 文本数据库等）库配合使用。

我在做文学程序，目前做法是数据库里面只保存文章结构，文章内容用一个个文件保存。

这样好处是数据库小了，查询快，不过全文搜索就不好办了。

我现在手上做的一个项目，这样处理。

形成分站结构，每个分站都对应一个相同结构的数据库，其中放 XXOO 张表，包括文章主题表。每个分站就等同于一个大类了。再加一个库（类似 xxoo_blog 这样的名字），目前一张表，保存发表人、时间、分站名、文章 ID 的对应记录，以做为集中区。

DB 封装类中这样定义基类：

[\[Copy to clipboard\]](#) [\[- \]](#)

CODE:

```
/******  
CLASS table base  
*****/  
class table {  
var $table_name;  
...  
  
function table()      {}  
//切换 DB  
function change_db($db_name)      {  
    global $db;  
    $db->Database=$db_name;  
    mysql_select_db($db_name,$db->Link_ID);  
}  
  
...  
}
```

相应表类定义：

[\[Copy to clipboard\]](#) [\[- \]](#)

CODE:

```
/******  
CLASS info  
*****/  
class info extends table{  
function info()      {  
    global $site_name;  
    $this->change_db($site_name);  
  
    $this->table_name="info";  
    $this->order_text="i_date desc";  
    $this->limit_text="";  
}
```

```
$this->id_name="i_id";  
}  
}
```

这样调用时只要注意好\$site_name 这个分站名的参数即可。

I 大型数据库的设计与编程技巧

本人最近开发一个访问统计系统，日志非常的大，都保存在数据库里面。

我现在按照常规的设计方法对表进行设计，已经出现了查询非常缓慢地情形。

大家对于这种情况如何来设计数据库呢？把一个表分成多个表么？
那么查询和插入数据库又有什么技巧呢？

谢谢，村里面的兄弟们！

按照时间把库分开，建立正确的索引，避免关联及子查询，like 的使用
给你两种方案：

数据表大了的话,肯定要分表的.

一种是按数据类型分表.

比如说用户 1 的日志,用户 2 的日志,查询的时候又基本上是按照这种方式查的话比较好.还省去了查询条件

另外一个方案就是按时间分:

其实来这也是一种类型,只是稍微有些差别.

如果每天的日志量很大的话可以当天写入临时表比如 log_tmp

然后每天定时跑 crontab,将 log_tmp 改名为 2007-03-05

然后重新建立一个 log_tmp 新表

另外也可以按多少条记录,分表,也可以考虑按 hash 算法分布表.

我个人认为可以按以下的思路优化一下：

1、大表，是指列数多还是行数多？

2、分表有按列分和按行分，

3、频繁操作是插入还是查询？占资源多的是那个，一般如果行数多会造成查询缓慢，

4、统计查询的时间实时要求高不高，比如是否一定要精确到某时某刻，还是某段时间（既可缓 10 分钟），如果可缓，可以 10 分钟做一次统计快照，既 10 分钟(或 5 分钟)做一次统计快照，把几千万数据统计为几万条数据的快照统计表，这样会明显提高效率。

还有，数据库系统主机的优化也很重要，比如，服务器分流，存储空间的设置技巧 ... 这些对大型数据库都很重要

I 方案探讨,关于工程中数据库的问题. [已结贴]

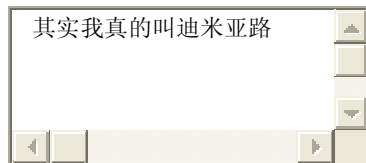
发表于：2007-06-01 09:17:26 楼主

用 VB 开发的工程,使用 SQL Server 数据库.

用 SQL Server 记录历史数据,有几个历史数据的表单,可是这样长年累月的添加记录,弄到这些表单庞大无比,以致对表单进行操作时消耗很长的时间.这个问题就显得很重要的了.

情况具体是这样,有 3 类表单,1 类是 1 分钟记录一次的表单,1 类是 20 分钟记录一次的表单,这个数据是 1 分钟表单的平均值记录表单.还有 1 类是 1 小时记录一次的表单,这个数据就是 20 分钟表单的平均值记录.

• Winters_lee



• 等级:

1 分钟记录一次的表单使用 1 个月就是

$1 \times 60 \times 24 \times 30 = 43200$ 条记录,这就有满大了,所以我限制了
这个表单的记录条数,在添加一条记录的时候就删除最老的一条,保证这个表单只有 1 个月的分量.

但是 20 分钟和 1 小时的表单我就不能这么做了,因为客户需要保留所有的记录.

曾试过用备份和恢复的方法,但是比较麻烦.请各位大侠看看有何良策,小弟不甚感激!

问题点数: 100 回复次数: 19

显示所有回

发表于: 2007-06-01 10:10:081 楼 得分:10

• [ZOU_SEAFARER](#)

颓废程序员^_^

但是 20 分钟和 1 小时的表单我就不能这么做了,因为客户需要保留所有的记录.

这个要求有点无理了,即便是最先进的系统,最大的磁盘空间也有满的那一天!!

• 等级:

• [cangwu_lee](#)

橙子

发表于: 2007-06-01 10:12:562 楼 得分:5

分表保存

• 等级:

发表于: 2007-06-01 10:22:103 楼 得分:0

• [Winters_lee](#)

其实我真的叫迪米亚路

楼上的,分表保存的话,问题来了:

需要在程序里面切入这个分开的表单名称,不然它不会知道该放到哪个表单里面去.然后还有如何定义不同的表单呢?这个在查询的时候同样也是比较麻烦的,查找时,必须判断要查找的数据是存放在哪个表单里面的.

• 等级:

发表于: 2007-06-01 10:38:234 楼 得分:10

• [vbman2003](#)

家人

从你的描述看,你这点数据量并不算大啊.就说你的分钟记录表,一个月才 43200,一年才 50 多万条数据,用 10 年也就 500 万数据,也不能称为庞大无比,如果这点数据,对于常规的操作,要消耗很长的时间,是不是硬件或者程序有问题?

• 等级:

发表于: 2007-06-01 11:00:355 楼 得分:0

• [Winters_lee](#)

其实我真的叫迪米亚路

50 多万条记录进行查询,求平均值等操作,你觉得会快的起来吗?

我在那机器上不开任何其他程序,就用 SQL Server 的查询分析器,使用这条命令:

```
select top 1 * from XXX order by InTime desc
```

大概用了我 8 秒的时间.而对数据记录比较少的表单来说,速度就比较快了,可以满足处理的要求.

• 等级:

• [Winters_lee](#)

发表于: 2007-06-01 11:04:306 楼 得分:0

其实我真的叫迪米亚路

尤其是更加精确的查询，耗时更多，象：

```
select * from XXX where InTime between "2003-01-01 " and "2003-05-30 "
```



等级：

发表于：2007-06-01 11:23:507 楼 得分:0

50 多万条记录进行查询，求平均值等操作，你觉得会快的起来吗？
我在那机器上不开任何其他的程序，就用 SQL Server 的查询分析器，使用这条命令：

vbman2003

家人

```
select top 1 * from XXX order by InTime desc
```

大概用了我 8 秒的时间。而对数据记录比较少的表单来说，速度就比较快了，可以满足处理的要求。



等级：

作为一台服务器，50 万数据的各种查询，返回数据都是毫秒级的
你的机器让人郁闷，这点数据，要在程序上或者数据库的设计上花费这样的精力，我真是无语

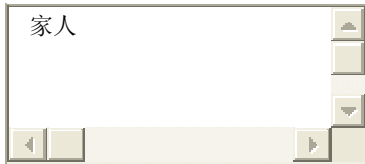
发表于：2007-06-01 11:28:478 楼 得分:0

50 多万条记录进行查询，求平均值等操作，你觉得会快的起来吗？

vbman2003

家人

我有个 access，其中的一张表有 70 万条数据，是从 SQL 数据库上备份下来的历史数据，我用 VB 连接查询一些统计信息，快的在 1 秒以内，最慢也不会超过 2 秒

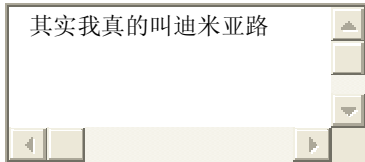


等级：

发表于：2007-06-01 13:32:319 楼 得分:0

数据库的操作对硬件的要求很高么？感觉对内存到是很有要求。

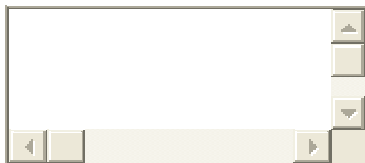
其实我真的叫迪米亚路



等级：

发表于：2007-06-01 14:03:3110 楼 得分:10

使用 sql server 的工作调度，写个存储过程，在每天适当的时候（如凌晨 3、4 点）建一新表，把昨天的数据都移到那个新表去，要统计的时候就用 union all 把需要的数据并在一起统计。。。然后定时做数据备份，清空旧的数据备份表。。以前我公司那套系统，平均每秒差不多都会有几十条新记录插入。。。如果一直放着不管它，早崩溃了。。。。

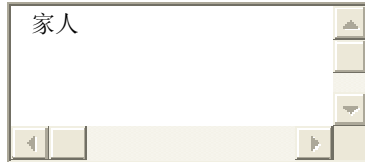


等级：

lsftest

发表于: 2007-06-01 14:21:27 11 楼 得分:0

• [vbman2003](#)



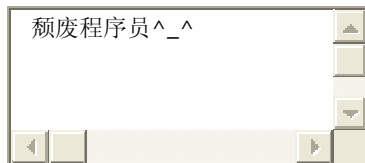
• 等级:

数据库的操作对硬件的要求很高么? 感觉对内存到是很有要求.

专业服务器从主板、CPU、内存、硬盘等等都与普通 PC 不一样的。

发表于: 2007-06-01 14:28:07 12 楼 得分:0

• [ZOU_SEAFARER](#)



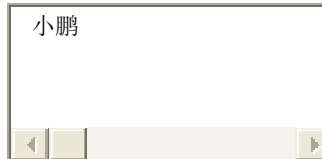
• 等级:

Isftest() 的方法我觉得不错,同时你还需要增加一个表,记录什么时候分表了,分表的名称等信息,到查询的时候就把这些表通过你增加的哪个新表联系起来!!

发表于: 2007-06-01 14:41:46 13 楼 得分:5

50 多万条记录进行查询,求平均值等操作,你觉得会快的起来吗? 我在那机器上不开任何其他程序,就用 SQL Server 的查询分析器,使用这条命令:

• [hupeng213](#)



• 等级:

```
select top 1 * from XXX order by InTime desc
```

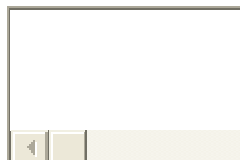
大概用了我 8 秒的时间. 而对数据记录比较少的表单来说,速度就比较快了,可以满足处理的要求.

这样子的现象只能证明一件事情, 你的数据库结构定义不合理, 适当的对某些字段建立索引,可以有效地提高速度。

发表于: 2007-06-01 20:36:20 14 楼 得分:10

同时你还需要增加一个表,记录什么时候分表了,分表的名称等信息,到查询的时候就把这些表通过你增加的哪个新表联系起来!!

• [Isftest](#)



• 等级:

=====

不需要,只要你自己心里有数就行了。。。。

例如,今天是 2007.06.01, 在 2007.06.02 的凌晨 3: 00, 工作调度就会执行预先写好的存储过程,大致要完成的工作是:

1.建新表,表名 testtable20070601, 表结构跟源表 (originaltable) 一样。

2.从源表中复制 2007.06.02 前的数据到 testtable20070601:

```
insert into testtable20070601 select * from original table where datefield < "2007-06-02 00:00:00.000 "
```

3.删除源表的旧数据:

```
delete from originaltable where datefield <
"2007-06-02 00:00:00.000 "
```

4.完成。

到了 2007.06.03 的凌晨 3: 00, 又重复执行上述操作, 只是那时建的新表表名是 **testtable20070602**。由于新表的表名都有规律, 要统计时只要找到需要的那些日期的表把它们 **union all** 再统计就行了。。。只是根据需要构建一个 **sql** 查询语句, 简单的字符串操作而已。。。

有需要的话, 另做一个存储过程, 也放在工作调度里。。作用是定时备份, 例如, 每个月的 1 号凌晨 4: 00 (为了不与上面 3: 00 那个冲突), 使用 **sql server** 的 **dts** 功能把 **testtable20070601**、**testtable20070602**、**testtable20070603** 等表导出为 **xls** 文件。存放于特定目录。然后再设一个时间判断, 每导出一个月的数据, 就把相隔几个月前的数据表删除。例如今天是 2007.6.1, 凌晨 3: 00 的时候就把 **testtable20070501**、**testtable20070502**.....**testtable20070531** 表导出到目录存放, 然后把 2007.04.01 前的备份表 **testtable20070301**、**testtable20070302** 从库里删除 (drop table? ? ? ?)。。。这时不把 **testtable20070401**~**testtable20070531** 也一并删除掉, 是预防在导出时出现问题而又删掉源数据就麻烦了。。。有一两个月的时间让你检查导出后的 **xls** 文件是否有问题, 总足够了把。。。另外一种做法是在工作调度中直接做备份, 把上面说的那些数据实时备份出来, 记得好像是一个 **mdb** 文件和一个 **ldf** 文件。

这种方法可以让你保留全部数据记录而又不会降低服务器效率及可以大量节省存储空间 (以前我把备份出来的数据文件用 **winrar** 最大压缩, 压缩比约为 10: 1)。。。


发表于: 2007-06-02 04:47:48 15 楼 得分:20

尤其是更加精确的查询, 耗时更多, 象:

```
select * from XXX where InTime between "2003-01-01 " and "2003-05-30 "
```

- [jiataizi](#)

佳太子

-  觉得还是楼主的数据库设计有问题, 象上面的这条语句, 即使数据有 100 万的, 如果数据库设计比较好的话, 处理也应该是毫秒级的,
- 等级: 建议楼主先试一下把 **InTime** 这个字段设置为聚集索引看看, 建议楼主看下这篇文章:

http://blog.csdn.net/great_domino/archive/2005/02/01/275839.aspx

发表于: 2007-06-02 13:06:2016 楼 得分:10

但是 20 分钟和 1 小时的表单我就不能这么做了,因为客户需要保留所有的记录.

既然用户只需要保留 20 分钟和 1 小时数据的所有记录,则 1 分钟的记录还用得着保存一个月吗? 只用 20 条就可以了。不会是一次性对一个月的 1 分钟数据进行统计才生成 20 分钟和 1 小时的数据吧, 那太低效了。

theforever



等级:

20 分钟和 1 小时的表单, 用户要求保留, 那就这样了。

但这样会产生两个头疼结果:

1. 硬盘容易满。

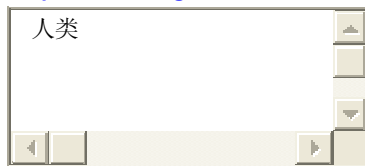
解决方法就是备份, 没啥说的。啥麻烦不麻烦, 用户要求不改变的情况下, 除了这个还能有什么办法? 何况这也不很麻烦。实际上任何象样的数据库应用软件都必须做好备份和恢复工作的, 这是基本。

2. 数据操作效率低。

既然量是没法压缩的, 就得讲究量的组织形式了。好的组织形式自然可以不受量的影响而致效率问题。合理地分表是少不了的。定期的备份清除也可能需要考虑, 那就看具体情况了。

发表于: 2007-06-02 21:59:5617 楼 得分:10

SupermanKing



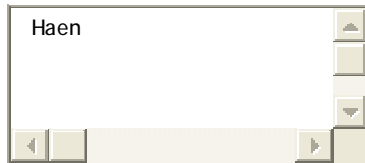
等级:

楼主可能没看过 ASP 海量数据查询的东西吧, 那么点数据要花那么久时间, 肯定是代码问题。

还有说到数据量和查询速度, SQL Server 2005 快很多, 不防看看, 我是深有体会, 但是几十万条的数据只要不超过 1G, Access 的数度都不慢, 何况 SQL Server

发表于: 2007-06-04 02:57:3018 楼 得分:10

haen_zhou



等级:

建议你采用分表记录联合查询...

建立分表的时候, 得采用有规律的表名称~~~

我记得以前我做过很多这样的生产数据记录, 都采用的分表...

其数据量比你的大得多, 1 分钟每个反应罐(有 20 个反应罐)要保存 3 条数据... 查询的时候都没有出现那样的问题...

I web 软件设计时考虑你的性能解决方案

关键字: 性能,WEB

前段时间搜罗了一些大型 web 应用程序开发的性能提升方案文章，但是一直不够系统。若现在让我设计一个支持大访问量的系统，仍然难于下手(以前没做过啊)

于是我把这些文章梳理了一下加入了自己的理解，记录了关键准则：

* 关键准则：

1. 选择什么编程语言不是问题
2. 选择的框架才可能影响系统的扩展和性能
3. 我倾向于以数据库为中心设计数据结构。
4. 分从两个方面提升性能：
 - 1). 软件设计方面
 - * 网页静态化
 - * 独立的图片服务器
 - * 可能采用中间缓存层服务器，最可能采用第三方成熟的软件
 - * 数据库分表(水平分割是最终方案)
 - 2). 系统、网络、硬件结构
 - * 集群：数据库集群，WEB 集群
 - * 采用：SAN
 - * 提升网络接入带宽

.....

其实，我最担心程序的设计架构问题成为制约将来系统扩展和性能提升时的因素。所以，这里也写出一个软件设计方面性能考虑的 Step By Step 实施方案供自己参考(而硬件扩展则可以根据并发用户数的升高随时调整)：

* Step By Step

假设采用 Java 语言作为主要开发语言，将 Tapestry + Spring + Hibernate + Mysql 作为基本架构。

阶段 I:

1. 以数据库为中心设计数据结构。最开始可以选择 Hibernate 作为 Persistency，如果需要切换（包括编程语言切换），这种设计思路会最大地减少移植障碍。
2. 基本的性能考虑：是否使用 OpenSessionView. 数据库设计一定的冗余度等。

阶段 II:

1. 网页静态化。
2. 独立的图片服务器。

阶段 III:

1. 中间缓存层组件的使用

阶段 IV:

1. 数据库分表：在软件设计上，我认为这几乎是提升性能的最后方法。

我认为每个阶段软件设计方面的修正，都将导致部分先期代码的更改，如果我们预先考虑到

网站的可能的设计方案更改，那么在软件代码实现的时候就会考虑到将来的修改，使将来的修改尽可能地少。

那么为什么我们一开始就让系统构架适应巨大并发量的访问呢？对于像我这样没有大型网站开发经验的人，或者还不确定系统的访问量会达到多大的前提下，又想尽快让网站上线，而且又不至于担心将来的扩展问题，那么我的做法未尝不是一个折衷呢？

草稿 2007-09-26

最后更新：2007-09-26 13:50

13:45 | [永久链接](#) | 浏览 (844) | [评论 \(6\)](#) | [收藏](#) | [进入论坛](#) | 发布在 Tapestry 圈子

[永久链接](#)

<http://koda.javaeye.com/blog/127276>

评论 共 6 条

[发表评论](#)

[wl95421](#) 2007-09-26 14:29

还是先想清楚你要做的网站对 session 的相关性有多大
能不能尽量将模块进行无关性分离
这样才是比较好的解决方案

如购物网站和论坛网站，对 session 的要求肯定不一致
架构也肯定不一样

[bluepoint](#) 2007-09-26 14:48

大体上这么玩可以，不过具体业务具体对待，这没有什么标准。

[timerri](#) 2007-09-26 15:31

影响性能的因素有哪些？其实只有下面几个方面：

1. 持久性数据查找速度
2. 持久性数据读写速度
3. 逻辑复杂度
4. 物理内存不够导致的虚拟存储频繁交换。

对应的解决方法：

1. 建立最合适的索引，建立缓存
2. 建立缓存，升级硬件
3. 精简，优化逻辑
4. 减少内存使用。

可以看出来，其实最需要做的，就是如何搞好缓存.....

为什么计算机界没有一个新职位，叫缓存工程师的？？

[ahuaxuan](#) 2007-09-26 18:08

说实话,我觉得楼主想了这么多还是没有抓住要领,任何一个软件,它的架构一定是在它的需求确定之后(指总体的业务需求,网站要达到的一个指标,包括业务特性),没有需

求就定架构是一种危险行为。楼主没有把自己网站性质,预期性能先确定就来谈用什么技术了,让人觉得有点空洞。

如果硬要给个方针,那么,应用集群+数据库集群就可以了

说到细节方面,第一个是 OpenSessionView 的问题,不会用 Hibernate 的人老是说 OpenSessionView 有问题, OpenSessionView 没有问题,说 OpenSessionView 有问题的人基本上都是用

Hibernate 用得有问题得人。

第二个是缓存,缓存可以加到很多层面,二级缓存和页面缓存得适用场景是不一样的,如果楼主在作架构的时候提到性能问题立刻就是中间层使用缓存,那么基本上可以说明楼主

对缓存的各种适用场景还不是非常了解,因为这些都是和业务相关(问题又回到了架构的确定需要在需求的确定之后)

第三在没有确定需求之前就一口咬定数据访问层是性能的瓶颈所在是站不住脚的。

那么在楼主现有的描述上,我也发表一下自己的看法:

1, 因为不是非常确定以后的访问量,那么为了便于扩展,应用在开发之初应该可以考虑使应用非常容易作集群部署(是农场,还是状态复制,如果是农场如何保证状态,是 cookie

, 还是 memcached, 还是用 blob 放到 db)

2, 在集群的环境下,使用如何使用缓存,哪些页面需要使用页面缓存 page cache, 哪些业务对象需要使用 Hibernate 的二级缓存等

我觉得楼主还是把网站得业务特性描述一下,这样才能更好得决定架构的设计。

Lucas Lee 2007-09-26 19:20

我认为目前还没有这种简单的框架能优雅的支持巨大访问量的。

为了高性能,总是有很多权衡的东西,需要额外的处理,想想 EJB 的机制吧,它就是为了高访问量设计的,但是不论访问量的大小一律都用它,则明显的使开发成本上升。

一般都会有这种多方面的权衡,ROR 在开发速度上的优势,是在损失了不少性能的前提下得到的,尽管它可能在中小访问量之下区别不算明显,但性能绝不会是它的优势。

koda 2007-09-26 19:37

ror 为什么会损失性能?能给出详细点的理由吗?

james2308 大哥,你那数据库分表的功能实现了吗

我发现你的汽车网数据很庞大,汽车的参数很全,汽车的种类也多,如果不分表的话运行会很慢很慢,请教你这个功能实现了没有,能不能说下思路或者直接分享呢[em23]

如果你的汽车的参数建在一个表里，那这个表即使分表了也会很慢，把参数建在多个表里，那分表也增加了难度，请教你如何实现的

这个是早就实现了的，应当是去年吧，就在我发出那个帖子的时间，实际上就已经实现了，

不好意思，一般情况下，只要我发出的一点观点和思想，我一般是首先要实现她，哪怕是简单的测试，只要通过，我才发布出来供大家参考

数据库分表的操作，就需要两点，我们从风讯自身的新闻来看，就是那个 News 表，我们就假定可以创建无数个 News1,news2。。。。。。这样的话，你就可以将你的数据到表数量到大一定量的时间(你自己规定)，就将主表，News 的数据转移到你的辅助表中，....因此从理论上将，只要我的硬盘足够大，我的站点，就可以运行 1000 年...并且不会影响生成的速度和运行速度

如果需要一个管理表来管理你可以任意增加的这些辅助表，这个管理表的主要作用，例如：如果你要获取某个栏目的新闻列表，有一种可能是这个栏目的新闻是在多个表中，这个就需要你调整程序来判断...

I 大型 Java Web 系统服务器选型问题探讨

作者: 佚名 出处:网络 (0)砖 (0)好 评论(0)条 [进入论坛](#)

更新时间: 2007-09-20 15:22

关 键 词: Java Web 服务器 系统选型

阅读提示: 如何能提高现有的基于 Java 的 Web 应用的服务能力呢? 由于架构模式和部署调优一直是 Java 社区的热门话题，这个问题引发了很多热心网友的讨论，其中一些意见对其它大型 Web 项目也有很好的指导意义。

一位网友在 JavaEye 询问了一个大型 Web 系统的架构和部署选型问题，希望能提高现有的基于 Java 的 Web 应用的服务能力。由于架构模式和部署调优一直是 Java 社区的热门话题，这个问题引发了很多热心网友的讨论，其中一些意见对其它大型 Web 项目也有很好的指导意义。在讨论之初 jackson1225 这样描述了当前的应用的架构和部署方案：

目前系统架构如下：

web 层采用 struts+tomcat 实现，整个系统采用 20 多台 web 服务器，其负载均衡采用硬件 F5 来实现；

中间层采用无状态会话 Bean+DAO+helper 类来实现，共 3 台 weblogic 服务器，部署有多个 EJB，其负载均衡也采用 F5 来实现；

数据库层的操作是自己写的通用类实现的，两台 ORACLE 数据库服务器，分别存放用户信息和业务数据；一台 SQL SERVER 数据库，是第三方的业务数据信息；

web 层调用 EJB 远程接口来访问中间件层。web 层首先通过一个 XML 配置文件中配置的 EJB 接口信息来调用相应的 EJB 远程接口；

该系统中一次操作涉及到两个 ORACLE 库以及一个 SQL SERVER 库的访问和操作，即有三个数据库连接，在一个事务中完成。

这样的架构其实很多公司都在使用，因为 Struts 和 Tomcat 分别是最流行的 Java Web MVC 框架和 Servlet 容器，而 F5 公司的负载均衡是横向扩展常见的解决方案（例如配置 session sticky 方案）。由于这个系统中有跨数据源的事务，所以使用 Weblogic Server EJB 容器和支持两阶段提交的数据库驱动就可以保证跨数据源的事物完整性（当然，容器管理的分布式事务并非是唯一和最优的解决方案）。

但是随着 Rod Johnson 重量级的著作《J2EE Development without EJB》和其中的 Spring 框架的流行，轻量级框架和轻量级容器的概念已经深入人心。所以对于 jackson1225 提出的这个场景，大多数网友都提出了置疑，认为这个系统滥用了技术，完全是在浪费钱。网友们大都认为 SLSB（无状态会话 Bean）完全没有必要出现在这个场景中，认为 SLSB 通过远程接口访问本地资源会有很大的性能开销，这种观点也是 Rod Johnson 在 without EJB 中批判 EJB 2.x 中的一大反模式。

由于 JavaEE 是一个以模式见长的解决方案，模式和架构在 JavaEE 中占有很重要的地位，所以很多业内专家也都警惕“反模式（Anti-patterns）”的出现。对于上面所述的方案是否是反模式，jackson1225 马上站出来申辩：

我们项目就是把 EJB 作为一个 Facade，只是提供给 WEB 层调用的远程接口，而且只用了无状态会话 Bean，所以性能上还可以的。

这个解释很快得到了一些网友的认可，但是大家很快意识到架构的好坏决定于是否能够满足用户的需求，davexin（可能是 jackson1225 的同事）描述了这个系统的用户和并发情况：

现在有用户 4000 万，马上要和另一个公司的会员系统合并，加起来一共有 9000 万用户。数据量单表中有一亿条以上的数据。这是基本的情况，其实我觉得现在的架构还是可以的，现在支持的并发大概 5000 并发用户左右，接下来会进行系统改造，目标支持 1 万个并发用户。

具体的并发量公布后又有网友质疑这个数据,认为这个系统的 Servlet 容器支持的并发数太小,怀疑是否配置不够优化。davexin 又补充了该项目的服务器配置:

系统前端 tomcat 都是用的刀片,配置在 2G 内存,cpu 大概在 2.0G,每台机器也就支持 250-400 个并发,再多的话,就会相应时间非常的常,超过 20 秒,失去了意义,所以我们才得出这样的结论的。

一位 ID 是 cauherk 的网友提出了比较中肯的意见,他没有从 Web 容器单纯的并发支持能力上提出改进方案,而是提出了对于类似的应用的一些通用的改进提示,这里摘要一下:

数据库压力问题

可以按照业务、区域等等特性对数据库进行配置,可以考虑分库、使用 rac、分区、分表等等策略,确保数据库能正常的进行交易。

事务问题

要在两个数据库中操作,那么必须考虑到分布式事务。你应该仔细的设计你的系统,来避免使用分布式事务,以避免分布式事务带来更多的数据库压力和其它问题。推荐你采用延迟提交的策略(并不保证数据的完整),来避免分布式事务的问题,毕竟 commit 失败的几率很低。

web 的优化

将静态、图片独立使用不同的服务器,对于常态的静态文件,采用 E-TAG 或者客户端缓存,google 很多就是这样干的。对于热点的功能,考虑使用完全装载到内存,保证绝对的响应速度,对于需要频繁访问的热点数据,采用集中缓存(多个可以采用负载均衡),减轻数据库的压力。

对于几乎除二进制文件,都应该在 L4 上配置基于硬件的压缩方案,减少网络的流量。提高用户使用的感知。

网络问题

可以考虑采用镜像、多路网络接入、基于 DNS 的负载均衡。如果有足够的投资,可以采用 CDN(内容分发网),减轻你的服务器压力。

cauherk 的这个分析比较到位,其中 ETags 的方案是最近的一个热点,InfoQ 的“使用 ETags 减少 Web 应用带宽和负载”里面对这种方案有很详细的介绍。一般以数据库为中心的 Web 应用的性能瓶颈都在数据库上,所以 cauherk 把数据库和事务问题放到了前两位来讨论。但是 davexin 解释在所讨论的这个项目中数据库并非瓶颈:

我们的压力不在数据库层，在 web 层和 F5。当高峰的时候，F5 也被点死了，就是每秒点击超过 30 万，web 动态部分根本承受不了。根据我们程序记录，20 台 web 最多承受 5000 个并发，如果再多，tomcat 就不响应了。就像死了一样。

这个回复让接下来的讨论都集中于 Web 容器的性能优化，但是 JavaEye 站长 robbin 发表了自己的意见，将话题引回了这个项目的架构本身：

performance tuning 最重要的就是定位瓶颈在哪里，以及瓶颈是怎么产生的。

我的推测是瓶颈还是出在 EJB 远程方法调用上！

tomcat 上面的 java 应用要通过 EJB 远程方法调用，来访问 weblogic 上面的无状态 SessionBean，这样的远程方法调用一般都在 100ms~500ms 级别，或者更多。而如果没有远程方法调用，即使大量采用 spring 的动态反射，一次完整的 web 请求处理在本地 JVM 内部的完成时间一般也不过 20ms 而已。一次 web 请求需要过长的执行时间，就会导致 servlet 线程被占用更多的时间，从而无法及时响应更多的后续请求。

如果这个推测是成立的话，那么我的建议就是既然你没有用到分布式事务，那么就干脆去掉 EJB。weblogic 也可以全部撤掉，业务层使用 spring 取代 EJB，不要搞分布式架构，在每个 tomcat 实例上面部署一个完整的分层结构。

另外在高并发情况下，apache 处理静态资源也很耗内存和 CPU，可以考虑用轻量级 web server 如 lighttpd/litespeed/nginx 取代之。

robbin 的推断得到了网友们的支持，davexin 也认同 robbin 的看法，但是他解释说公司认为放弃 SLSB 存在风险，所以公司倾向于通过将 Tomcat 替换为 Weblogic Server 10 来提升系统的用户支撑能力。robbin 则马上批评了这种做法：

坦白说我还从来没有听说过大规模互联网应用使用 EJB 的先例。为什么大规模互联网应用不能用 EJB，其实就是因为 EJB 性能太差，用了 EJB 几乎必然出现性能障碍。

web 容器的性能说到底无非就是 Servlet 线程调度能力而已，Tomcat 不像 WebLogic 那样附加 n 多管理功能，跑得快很正常。对比测试一下 WebLogic 的数据库连接池和 C3PO 连接池的性能也会发现类似的结论，C3PO 可要比 WebLogic 的连接池快好几倍了。这不是说 WebLogic 性能不好，只不过 weblogic 要实现更多的功能，所以在单一的速度方面就会牺牲很多东西。

以我的经验来判断，使用 tomcat5.5 以上的版本，配置 apr 支持，进行必要的 tuning，使用 BEA JRockit JVM 的话，在你们目前的刀片上面，支撑 500 个并发完全是可以做到的。结合你们目前 20 个刀片的硬件，那么达到 1 万并发是没问题的。当然这样做的前提是必须扔掉 EJB，并置 web 层和业务层在同一个 JVM 内部。

接下来 robbin 还针对 davexin 对话题中的应用分别在 tomcat 和 weblogic 上的测试数据进行了分析：

引用：

2。1 台 weblogic10 Express（相当于 1 台 tomcat，用于发布 jsp 应用）加 1 台 weblogic10（发布 ejb 应用），能支持 1000 个并发用户.....

.....

4。1 台 tomcat4.1 加 1 台 weblogic8，只能支持 350 个并发用户，tomcat 就连结超时，说明此种结构瓶颈在 tomcat。

这说明瓶颈还不在 EJB 远程调用上，但是问题已经逐渐清楚了。为什么 weblogic 充当 web 容器发起远程 EJB 调用的时候可以支撑 1000 个并发，但是 tomcat 只能到 350 个？只有两个可能的原因：

你的 tomcat 没有配置好，严重影响了性能表现

tomcat 和 weblogic 之间的接口出了问题

接着 springside 项目发起者江南白衣也提出了一个总体的优化指导：

1. 基础配置优化

tomcat 6? tomcat 参数调优?

JRockit JVM? JVM 参数调优?

Apache+Squid 处理静态内容?

2. 业务层优化

部分功能本地化，而不调 remote session bean?

异步提交操作, JMS?

cache 热点数据?

3. 展示层优化

动态页面发布为静态页面?

Cache 部分动态页面内容?

davexin 在调整了 Tomcat 配置后应验了 robbin 对 tomcat 配置问题的质疑，davexin 这样描述经过配置优化以后的测试结果：

经过测试，并发人数是可以达到像 robbin 所说的一样，能够在 600 人左右，如果压到并发 700 人，就有 15% 左右的失败，虽然在调整上面参数之后，并发人数上去了，但是在同样的时间内所完成的事务数量下降了 10% 左右，并且响应时间延迟了 1 秒左右，但从整体上来说，牺牲一点事务吞吐量和响应时间，并发人数能够提高 500，觉得还是值得的。

至此这个话题有了一个比较好的结果。这个话题并非完全针对一个具体的项目才有意义，更重要的是在分析和讨论问题的过程中网友们解决问题的思路，尤其是 cauherk、robbin、江南白衣等几位网友提出的意见可以让广大 Java Web 项目开发者了解到中、大型项目所需要考虑的架构和部署所需要考虑的关键问题，也消除了很多人对轻量 Servlet 容器与 EJB 容器性能的一些误解。

富有挑战性的问题,建立超大数据库的问题.

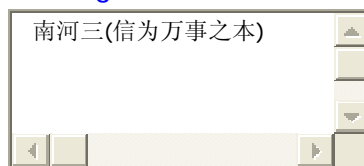
发表于: 2007-03-28 09:32:11 楼主

现在要设计一个记录数非常大的数据库表,表的结构非常简单,但记录数会非常大,可能会有几十亿条,但表的字段只包括几个数字列,再有一个列用来保存图片,我现在准备采用 BFILE 类型来把图片路径保存在表中,另外表会根据某个列数据进行分区.

在设计这样的大表还应该注意哪些问题..欢迎大家赐教.

凡提出较好意见的,可以另行开贴再加分.

- [haiwangstar](#)

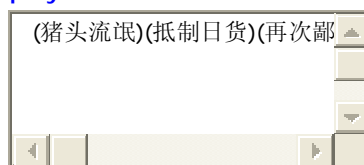


-
- 等级:

问题点数: 100 回复次数: 39

显示所有回

- [playmud](#)

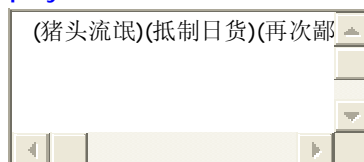


-
- 等级:

发表于: 2007-03-28 09:44:151 楼 得分:0

建议分表操作,都放入一个表内会严重影响速度.可以按照类型分,可以按照时间分.
总之坚决避免大表的出现.

- [playmud](#)



-
- 等级:

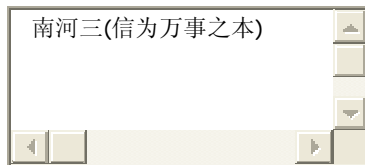
发表于: 2007-03-28 09:46:162 楼 得分:0

如果你不听劝告,任性而为,那就把需要查询的项或者组合做索引,合理的给这个表分配物理空间.

发表于: 2007-03-28 09:47:253 楼 得分:0

谢谢楼上的朋友,表分区是肯定的,我上面也写了.

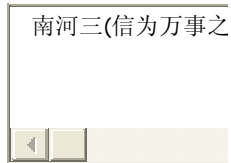
- [haiwangstar](#)



等级:

发表于: 2007-03-28 10:01:554 楼 得分:0

haiwangstar



等级:

另外关于索引, 表会有 3 个数字列, 一个是级别, 共 15 级, 另一个是 X, 再有一个是 Y. 这两列的数据范围都是从 0 到 1000 左右, 查询的时候每次都会用到这三个列, 大抵应该是这样

```
select image from table1 WHERE level = ? and  
x = ? and y = ?
```

这个时候如何建索引会比较好, 将级别建为簇索引, X, Y 建为复合索引? 还是三个列统一建为复合索引好..

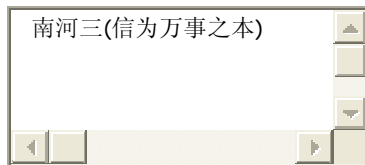
发表于: 2007-03-28 10:47:105 楼 得分:0

另外还有 IO 均衡的问题, 另一个设计者在看了上面的方案后, 认为如果这样做的话, 网络 IO 会集中在中一台机器上, 形成瓶颈. 在群集的情况下也会这样的吗?

他是想得到的图片的路径后, 直接去那台服务器去读图.

另外, 对于这样的超大数据库, 磁盘是一个整体的阵列, 还是每个服务器有自己的磁盘阵列呢? 这个问题可能有点弱.. 我过去也没做过这么大的数据库. 还望大家能多多赐教.

haiwangstar



等级:

发表于: 2007-03-29 08:53:016 楼 得分:0

junqiang



等级:

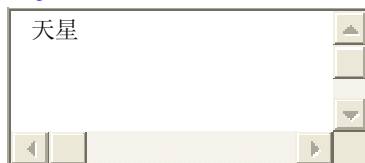
没这方面的经验, 只是理论:

rac 集群一般是共享磁盘组, 一般来说磁盘组的 io 性能很好, 带宽高 (高级的是光纤连接)。

rac 集群的网络 io 不会集中在一台服务器上, 会自动负载平衡。

发表于: 2007-03-29 09:42:587 楼 得分:0

skystar99047



等级:

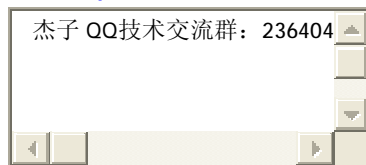
分区数目可以考虑增大。

每个区的表空间可以考虑放在不同的物理空间上。

分区索引是必须的。如果增删改频率较低, 查询较多, 可以考虑位图索引。

如果图片保存在表中, 需要考虑将该字段的存储放在另一单独的物理空间上。

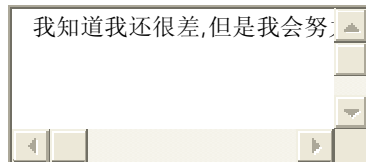
- [i_love_pc](#) 发表于：2007-03-29 11:11:228 楼 得分:0



几十亿条
=====
的确有点多

- 等级:

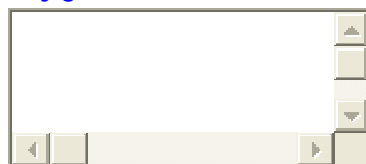
- [renjun24](#) 发表于：2007-03-29 11:19:009 楼 得分:0



up

- 等级:

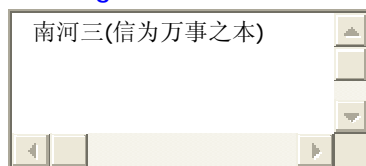
- [huylgghost](#) 发表于：2007-03-29 11:30:2710 楼 得分:0



几十亿条, level , x, y, 图片,
google earth 是不是就是这么做的?

- 等级:

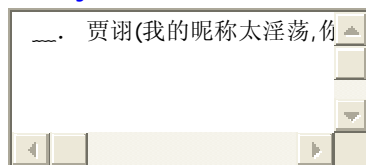
- [haiwangstar](#) 发表于：2007-03-29 12:03:3211 楼 得分:0



楼上的朋友,没错.就是做 EARTH MAP

- 等级:

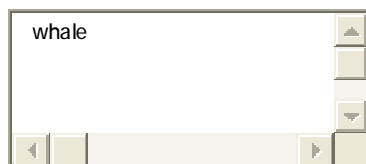
- [lin_style](#) 发表于：2007-03-29 12:08:3212 楼 得分:0



这样的话。
扩充什么就不要考虑了。
设计个最适合查询的。

- 等级:

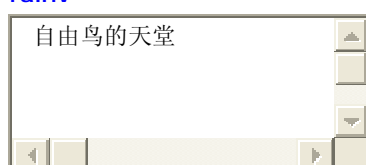
- [whalefish2001](#) 发表于：2007-03-29 12:40:2113 楼 得分:0



索引是必要的，不过，索引会占用很大空间。

- 等级:

- [rainv](#) 发表于：2007-03-29 13:29:3514 楼 得分:0

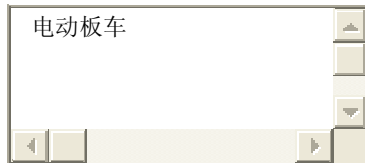


mark!
没接触过这种项目.^-^

-

• 等级:

• [e_board](#)

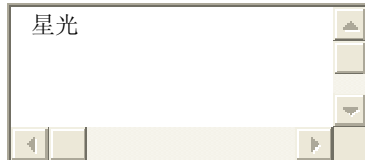


发表于: 2007-03-29 13:34:2815 楼 得分:0

MySQL 中有分区表的概念;MySQL 会自动处理这些,不知道 Oracle 有没有类似的

• 等级:

• [yanxinhao972](#)

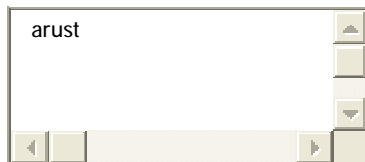


发表于: 2007-03-29 13:37:4816 楼 得分:0

ORACLE 10g 中可以创建分区表

• 等级:

• [arust](#)

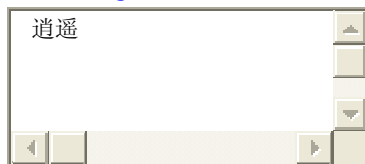


发表于: 2007-03-29 14:35:1817 楼 得分:0

这种数据库用 PostgreSQL 比较好

• 等级:

• [thinkinnight](#)

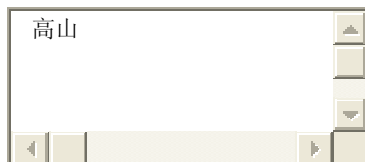


发表于: 2007-03-29 15:43:1418 楼 得分:0

不错, 学习

• 等级:

• [conanfans](#)

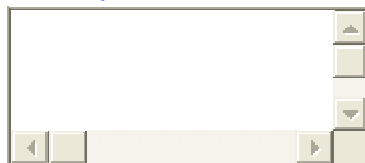


发表于: 2007-03-29 16:28:5619 楼 得分:0

ORACLE 在大数据量上不如 DB2

• 等级:

• [smallSophia](#)

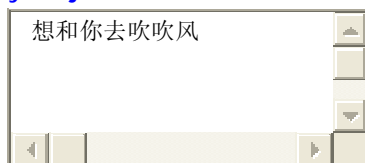


发表于: 2007-03-29 16:42:0120 楼 得分:0

我只能说学习, 继续关注!

• 等级:

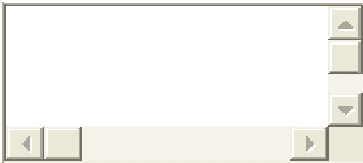
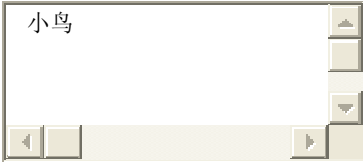
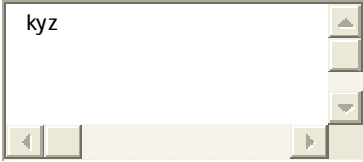
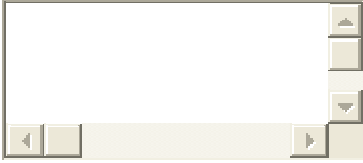

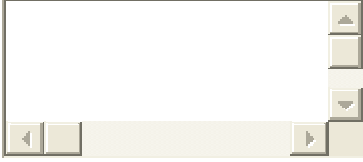

• [yxsalj](#)

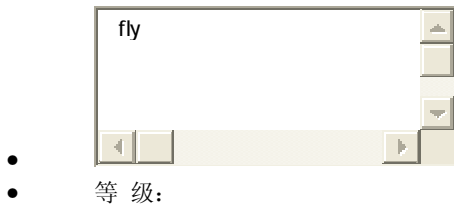


发表于: 2007-03-29 17:25:1821 楼 得分:0

几十亿也不是很多,分区,加上合适的索引,问题也不大

• 等级:

- [murphyding](#)
 发表于：2007-03-29 17:28:0522 楼 得分:0
初次登陆，向大家问好，哈哈
等级:
- [prcgolf](#)
 发表于：2007-03-29 18:02:2423 楼 得分:0
up
等级:
- [winesmoke](#)
 发表于：2007-03-29 18:23:2024 楼 得分:0
那位高手还是整个方案出来噻！
关注！
等级:
- [MONOLINUX](#)
 发表于：2007-03-29 21:54:2125 楼 得分:0
该回复于 2007-10-09 14:24:38 被管理员删除
等级:
- [dbpointer](#)
 发表于：2007-03-29 22:30:4826 楼 得分:0
楼上的牛啊，不过搜索面好像还不如百度
等级:
- [uniume](#)
 发表于：2007-03-29 22:40:3227 楼 得分:0
该回复于 2007-10-14 16:54:02 被管理员或版主删除
等级:
- [kkk_visual](#)
 发表于：2007-03-29 23:14:3728 楼 得分:0
帮顶一下。
等级:
- [flyycyu](#)
发表于：2007-03-30 00:20:4629 楼 得分:0



-
- 等级:

目前正做完这么一个系统,和你的类似,数据大概是每个表 9 亿左右,但是有 6,7 个表都是这么大数据量,一个表的字段大概有 50,60 个,主要都是 float.

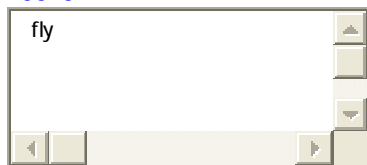
不知道你的数据是怎么进入的,我们系统对数据的装入也是有要求的.

第一,个就是分表操作,在以前用 infomix 之类的系统的开发都采用,我在做我们系统的时候第一个方案设计出来的就是分表,当然带来的问题是维护问题,系统中上万个表,基本上图形控制台是打不开,所以最好的方式是分表+分区,oracle 专家也是这么建议的!

第二,簇索引我没有怎么用过,但是如果你经常 3 个组合查,就建复合索引,或者在类别上建建 BITMAP 索引,对 x,y 建复合索引,另外,我觉得呢,如果按我们现在开发系统的经验来说,你应该把类别做为分表,这样在类别上就不存在建立索引问题,而对 x,y,按某种方式在表内进行分区

发表于: 2007-03-30 00:27:0130 楼 得分:0

- flyycyu



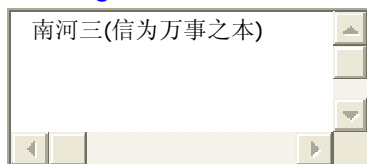
-
- 等级:

另外,如果可能的话,用 BFILE,还不如系统放在文件系统上,而数据库只做连接,当然这个看你,用 jdbc 插入当然会比直接拷贝文件系统慢.不过可能管理上带来方便性.另外 io 均衡问题你不用考虑,在建立数据库时候,这个表,或者是数据文件直接写文件系统的话,你把表空间或者文件系统挂在裸设备上,而不要用本地文件系统,系统会自动给你处理均衡问题的!

发表于: 2007-03-30 09:19:4031 楼 得分:0

flyycyu(fly) 这位朋友,非常感谢您的意见!!

- haiwangstar

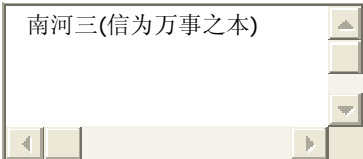


-
- 等级:

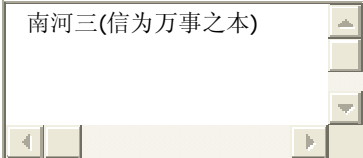
我过去从来没有搞过这么大的数据库系统,所以我上面所讲述的一切都只是纸上谈兵.但我的思路,设想几乎同你都是一致的.关于是否用 BFILE 的问题,这个也是一个不大关乎全局的小问题.

我只所以特意问会不会有 IO 不均衡问题,是因为我的同事认为我们这样的方案是不行的,IO 会不均衡(他认为还有很多问题).而我认为绝对不会出现这种情况的,因为 ORACLE 在设计中他不可能不考虑这样显而易见的问题,我也查了资料,在 ORACLE 集群中,网络负荷也是存在 ORACLE 负荷均衡的.

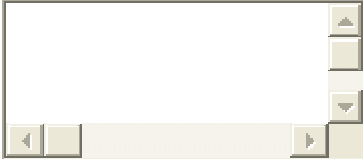
看来这样的方案才是经实践检验过的可行的.

- [haiwangstar](#) 发表于: 2007-03-30 09:24:2932 楼 得分:0
南河三(信为万事之本)
等级:

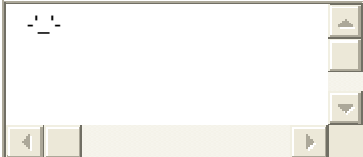
另外还有一个问题,就是 [flycyu\(fly\)](#) 这位朋友 你们是使用 **SAN** 存储设备的吗,光纤网络? 这套系统是不是非常昂贵? 即使不是,我想也一定是共享统一存储器吧.

- [haiwangstar](#) 发表于: 2007-03-30 09:28:1333 楼 得分:0
南河三(信为万事之本)
等级:

我们系统对数据的装入也是有要求的.
朋友这句话是怎么讲.能说一下吗

- [asker100](#) 发表于: 2007-03-30 09:54:4034 楼 得分:0

等级:

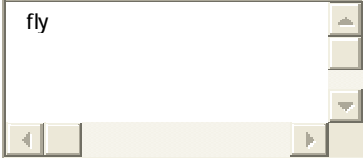
这种情况就不要用数据库了, 直接的文件存储+优化的索引, 要看穿数据库这种东西

- [gameboy999](#) 发表于: 2007-03-30 10:41:1835 楼 得分:0
._.
等级:

同意楼上, 地图数据自己的特点, 不一定需要通用的数据库

- [wuluhua2003](#) 发表于: 2007-03-30 15:53:5636 楼 得分:0
人人为我,我为人人
等级:

学习了

- [flycyu](#) 发表于: 2007-03-31 09:55:4937 楼 得分:0
fly
等级:

对, **san**, 光纤网络, 客户有钱, 而且事情又重要.
装入数据的要求是我们数据是批量装入的, 一年集中在 **1, 2** 个时间点, 平时就是查询, 而装入数据时候, 系统上会有几十个并发解析 **10 万-30 万** 之间的数据包.
至于 **BFILE** 问题, 我只是建议, 也可能是自己学艺不精, 因为我的数据文件一般在 **10m** 以上, 所以在上亿后, 存库老是出些莫名其妙的问题, 所以最后干脆就存外部文件, 至少在过程上, 你少了一道由 **web** 服务器把数据包发包到数据库服务器的过程.

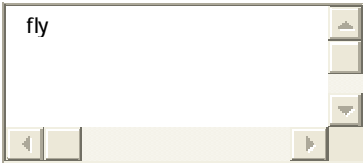
还有 io 均衡这些问题，我觉得一是自己调很难，加大难度和时间，也未必出来后最优，还不如利用硬件设备，还有像 10g 里面的 ASM。我们的数据量像上面所说，在
ibm 570,8cpu,32g 内存下，并发解析数据包到 40, 50 都没问题，时间在 50 秒以内都能完成 10 万数据装入，而查询这些就更不用说了，压力测试上 800 都没问题，当然这是非集群环境，当然你的具体业务还是由你分析，这些只是建议

发表于：2007-03-31 10:10:0438 楼 得分:0

再说下 io 均衡问题，我只是说下实际部署中碰到的问题，因为这个只有具体问题具体分析，一个是磁盘的 io,因为在设计 TABLESPACE 时候，你已经有意识的根据业务划分到不同的磁盘块上面了，我当时在测试上碰到的问题就是对回滚段的资源占用也很大，后来回滚的表空间分到 5 个磁盘上去了，性能马上就好了上来。

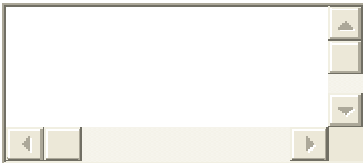
另外一个是网络 io,我看你提的是网络 io,而不是磁盘 io，不清楚你的这个网络 io 指得是客户请求到 web 服务器，还是 web 服务器到数据库服务器！因为我们业务的关系，oracle 没有架集群，当然也是还用不到那功能，所以没有参考意见，至于 web 请求这块，我觉得解决这个方案应该很多吧？比如我们，最后是按照业务模块来划分的多台 web 服务器.当然我们系统特定和你的可能有一定程度类似，就是大部分时间主要是查询。

• flyycyu



等级:

• huylgghost



等级:

发表于：2007-04-05 10:13:2339 楼 得分:0

进来学习一下

发表于：2007-04-06 11:07:4040 楼 得分:0

skystar99047(天星

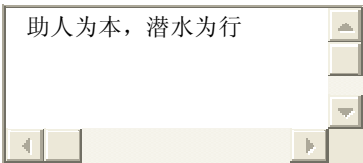
分区数目可以考虑增大。

每个区的表空间可以考虑放在不同的物理空间上。

分区索引是必须的。如果增删改频率较低，查询较多，可以考虑位图索引。

如果图片保存在表中，需要考虑将该字段的存储放在另一单独的物理空间上。

• hrui99



等级:

同意上面描述。补充 SELECT 描述考虑加入 HINTS 描述

--并行处理

如: `select /*+ parallel(tab,处理器个数) */`

高并发高流量网站架构

Architecture of Website with High Page view and High concurrency

院系：信息科学学院

专业：计算机科学与技术

学号：03281077

姓名：唐福林

指导教师：朱小明

北京师范大学

2007 年 3 月

北京师范大学士学位论文（设计）原创性声明

本人郑重声明： 所呈交的学士学位论文（设计），是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引

本人签名: _____ 年 月 日

保密论文在 年解密后适用本授权书。

导师签名： 年 月 日

摘要

Web2.0 的兴起，掀起了互联网新一轮的网络创业大潮。以用户为导向的新网站建设概念，细分了网站功能和用户群，不仅成功的造就了一大批新生的网站，也极大的方便了上网的人们。但 Web2.0 以用户为导向的理念，使得新生的网站有了新的特点——高并发，高流量，数据量大，逻辑复杂等，对网站建设也提出了新的要求。

本文围绕高并发高流量的网站架构设计问题，主要研究讨论了以下内容：

首先在整个网络的高度讨论了使用镜像网站，CDN 内容分发网络等技术对负载均衡带来的便利及各自的优缺点比较。然后在局域网层次对第四层交换技术，包括硬件解决方案 F5 和软件解决方案 LVS，进行了简单的讨论。接下来在单服务器层次，本文着重讨论了单台服务器的 Socket 优化，硬盘级缓存技术，内存级缓存技术，CPU 与 IO 平衡技术（即以运算为主的程序与以数据读写为主的程序搭配部署），读写分离技术等。在应用层，本文介绍了一些大型网站常用的技术，以及选择使用该技术的理由。最后，在架构的高度讨论了网站扩容，容错等问题。

本文以理论与实践相结合的形式，结合作者实际工作中得到的经验，具有较广泛的适用性。

关键词：高并发 高流量 网站架构 网站扩容 容错

Abstract

With web2.0 starting, raised the Internet new turn of network to start undertaking the flood tide. To be user-oriented concept of the new websites, not only successfully created a large number of new sites, but also greatly facilitate the development of the Internet people. The Web2.0 at the same time take the user as the guidance idea, enabled the newborn website to have the new characteristic - high concurrency, high page views, big data quantity, and complex logic, etc., also set the new request to the website construction. This article revolves the high concurrent high current capacity the website overhead construction design question, the main research discussed these content: First discussed the usage of mirror sites in the entire network, CDN content distribution network, the convenience and respective good and bad points comparison which brings to the load balancing. Then in the local area network, the fourth level exchange technology, including hardware solution F5 and software solution LVS, has been carried on with a simple discussion. Received in the single server level, this article emphatically discussed the single server socket optimization, the hard disk cache technology, the memory level buffer technology, CPU and the IO balance technology, the read-write separation technology and so on. In the application level, this article introduced some large-scale website commonly used technologies, as well as the reason of choice of these technicals. Finally, highly discussed the website in the overhead construction to expand accommodates, fault-tolerant. This article form which unifies by the theory and the practice, experience which in the author practical work obtains, has amore widespread serviceability.

KEY WORDS: high page view, high concurrency, architecture of website site, expansion

目 录

1 引言 9

- 1.1 互联网的发展 9
- 1.2 互联网网站建设的新趋势 9
- 1.3 新浪播客的简介 11

2 网络层架构 12

- 2.1 镜像网站技术 12
- 2.2 CDN 内容分发网络 13
- 2.3 应用层分布式设计 16
- 2.4 网络层架构小结 17

3 交换层架构 17

- 3.1 第四层交换简介 17
- 3.2 硬件实现 18
- 3.3 软件实现 18

4 服务器优化 19

- 4.1 服务器整体性能考虑 19
- 4.2 Socket 优化 19
- 4.3 硬盘级缓存 22
- 4.4 内存级缓存 24
- 4.5 CPU 与 IO 均衡 26
- 4.6 读写分离 26

5 应用程序层优化 28

- 5.1 网站服务器程序的选择 28
- 5.2 数据库选择 29

5.3 服务器端脚本解析器的选择 30

5.4 可配置性 32

5.5 封装和中间层思想 33

6 扩容、容错处理 33

6.1 扩容 33

6.2 容错 34

7 总结及展望 35

7.1 总结 35

7.2 展望 36

I 高并发高流量网站架构

1 引言

1.1 互联网的发展

最近十年间，互联网已经从一个单纯的用于科研的，用来传递静态文档的美国内部网络，发展成了一个应用于各行各业的，传送着海量多媒体及动态信息的全球网络。从规模上看，互联网在主机数、带宽、上网人数等方面几乎一直保持着指数增长的趋势，2006 年 7 月，互联网上共有主机 439, 286, 364 台，WWW 站点数量达到 96, 854, 877 个 [1]。全球上网人口在 2004 年达到 7 亿 2900 万 [2]，中国的上网人数在 2006 年 12 月达到了约 1 亿 3700 万 [3]。另一方面，互联网所传递的内容也发生了巨大的变化，早期互联网以静态、文本的公共信息为主要内容，而目前的互联网则传递着大量的动态、多媒体及人性化的信息，人们不仅可以通过互联网阅读到动态生成的信息，而且可以通过它使用电子商务、即时通信、网上游戏等交互性很强的服务。因此，可以说互联网已经不再仅仅是一个信息共享网络，而已经成为了一个无所不在的交互式服务的平台。

1.2 互联网网站建设的新趋势

互联网不断扩大的规模，日益增长的用户群，以及 web2.0 [4] 的兴起，对互联网网站建设提出了新的要求：

- 高性能和高可扩展性。2000 年 5 月，访问量排名世界第一（统计数据来源于 [5]）的 Yahoo [6] 声称其日页浏览数达到 6 亿 2500 万，即每秒

约 30,000 次 HTTP 请求(按每个页面浏览平均产生 4 次请求计算)。这样大规模的访问量对服务的性能提出了非常高的要求。更为重要的是,互联网受众的广泛性,使得成功的互联网服务的访问量增长潜力和速度非常大,因此服务系统必须具有非常好的可扩展性,以应付将来可能的服务增长。

- 支持高度并发的访问。高度并发的访问对服务的存储与并发能力提出了很高的要求,当前主流的超标量和超流水线处理器能处理的并发请求数是有限的,因为随着并发数的上升,进程调度的开销会很快上升。互联网广域网的本质决定了其访问的延迟时间较长,因此一个请求完成时间也较长,按从请求产生到页面下载完成 3 秒计算, Yahoo 在 2000 年 5 月时平均有 90,000 个并发请求。而且对于较复杂的服务,服务器往往要维护用户会话的信息,例如一个互联网网站如果每天有 100 万次用户会话,每次 20 分钟的话,那平均同时就会有约 14000 个并发会话。
- 高可用性。互联网服务的全球性决定了其每天 24 小时都会有用户访问,因此任何服务的停止都会对用户造成影响。而对于电子商务等应用,暂时的服务中止则意味着客户的永久失去及大量的经济损失,例如 ebay.com [7] 1999 年 6 月的一次 22 小时的网站不可访问,对此网站的 380 万用户的忠诚度造成巨大影响,使得 Ebay 公司不得不支付了近 500 万美元用于补偿客户的损失,而该公司的市值同期下降了 40 亿美元 [8]。因此,关键互联网应用的可用性要求非常高。

1.3 新浪播客的简介

以 YouTube [9] 为代表的微视频分享网站近来方兴未艾,仅 2006 年一年,国内就出现近百家仿 YouTube 的微视频分享网站 [10],试图复制 YouTube 的成功模式。此类网站可以说是 Web2.0 概念下的代表网站,具有 Web2.0 网站所有典型特征:高并发,高流量,数据量大,逻辑复杂,用户分散等等。新浪 [11] 作为国内最大的门户网站,在 2005 年成功运作新浪博客的基础上,于 2006 年底推出了新浪播客服务。新浪播客作为国内门户网站中第一个微视频分享服务的网站,依靠新浪网站及新浪博客的巨大人气资源,在推出后不到半年的时间里,取得了巨大的成功:同类网站中上传视频数量第一、流量增长最快、用户数最多 [12],所有这些成绩的取得的背后,是巨大的硬件投入,良好的架构支撑和灵活的应用层软件设计。

本文是作者在新浪爱问搜索部门实习及参与新浪播客开发的经验和教训的回顾,是作者对一般高并发高流量网站架构的总结和抽象。

2 网络层架构

2.1 镜像网站技术

镜像网站是指将一个完全相同的站点放到几个服务器上,分别有自己的 URL,这些服务器上的网站互相称为镜像网站 [13]。镜像网站和主站并

没有太大差别，或者可以视为主站的拷贝。镜像网站的好处是：如果不能对主站作正常访问（如服务器故障，网络故障或者网速太慢等），仍能通过镜像服务器获得服务。不便之处是：更新网站内容的时候，需要同时更新多个服务器；需要用户记忆超过一个网址，或需要用户选择访问多个镜像网站中的一个，而用户选择的，不一定是最优的。在用户选择的过程中，缺乏必要的可控性。

在互联网发展的初期，互联网上的网站内容很少，而且大都是静态内容，更新频率底。但因为服务器运算能力低，带宽小，网速慢，热门网站的访问压力还是很大。镜像网站技术在这种情况下作为一种有效解决方案，被广泛采用。随着互联网的发展，越来越多的网站使用服务器端脚本动态生成内容，同步更新越来越困难，对可控性要求越来越高，镜像技术因为不能满足这类网站的需要，渐渐的淡出了人们的视线。但有一些大型的软件下载站，因为符合镜像网站的条件——下载的内容是静态的，更新频率较低，对带宽，速度要求又比较高，如国外的 SourceForge（<http://www.SourceForge.net>，著名开源软件托管网站），Fedora（<http://fedoraproject.org>，RedHat 赞助的 Linux 发行版），国内的华军软件园（<http://www.onlinedown.net>），天空软件站（<http://www.skycn.com>）等，还在使用这项技术（图 1）。



图 1 上图：
天空软件站
首页的镜像
选择页面
下图：



SourceForge 下载时的镜像选择页面

在网站建设的过程中，可以根据实际情况，将静态内容作一些镜像，以加快访问速度，提升用户体验。

2.2 CDN 内容分发网络

CDN 的全称是 Content Delivery Network，即内容分发网络。其目的是通过在现有的互联网中增加一层新的网络架构，将网站的内容发布到最接近用户的网络“边缘”，使用户可以就近取得所需的内容，分散服务器的压力，解决互联网拥挤的状况，提高用户访问网站的响应速度。从而解决由于网络带宽小、用户访问量大、网点分布不均等原因所造成的用户访问网站响应速度慢的问题 [14]。

CDN 与镜像网站技术的不同之处在于网站代替用户去选择最优的内容服务器，增强了可控制性。CDN 其实是夹在网页浏览者和被访问的服务器中间的一层镜像或者说缓存，浏览者访问时点击的还是服务器原来的 URL 地址，但是看到的内容其实是对浏览者来说最优的一台镜像服务器上的页面缓存内容。这是通过调整服务器的域名解析来实现的。使用 CDN 技术的域名解析服务器需要维护一个镜像服务器列表和一份来访 IP 到镜像服务器的对应表。当一个用户的请求到来的时候，根据用户的 IP，查询对应表，得到最优的镜像服务器的 IP 地址，返回给用户。这里的最优，需要综合考虑服务器的处理能力，带宽，离访问者的距离远近等因素。当某个地方的镜像网站流量过大，带宽消耗过快，或者出现服务器，网络等故障的时候，可以很方便的设置将用户的访问转到另外一个地方（图 2）。这样就增强了可控制性。

图 2



CDN 原理示意图

CDN 网络加速技术也有它的局限性。首先，因为内容更新的时候，需要同步更新多台镜像服务器，所以它也只适用于内容更新不太频繁，或者对实时性要求不是很高的网站；其次，DNS 解析有缓存，当某一个镜像网站的访问需要转移时，主 DNS 服务器更改了 IP 解析结果，但各地的 DNS 服务器缓存更新会滞后一段时间，这段时间内用户的访问仍然会指向该服务器，可控制性依然有不足。

目前，国内访问量较高的大型网站如新浪、网易等的资讯频道，均使用 CDN 网络加速技术（图 3），虽然网站的访问量巨大，但无论在什么地方访问，速度都会很快。但论坛，邮箱等更新频繁，实时性要求高的频道，则不适合使用这种技术。

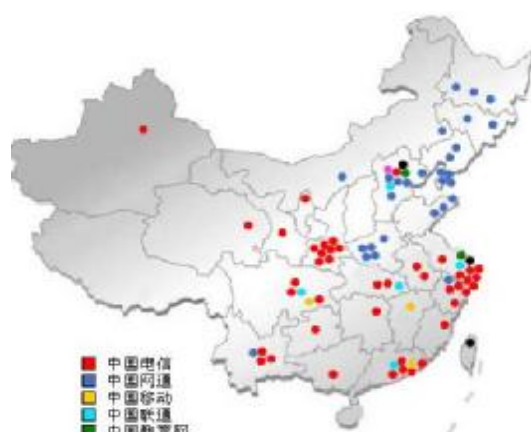


图 3 新浪网使用 Chi naCache CDN 服务。

Chi naCache 的服务节点全球超过 130 个，

其中中国节点超过 80 个，

覆盖全国主要 6 大网络的主要省份 [15] 。

2.3 应用层分布式设计

新浪播客为了获得 CDN 网络加速的优点，又必须避免 CDN 的不足，在应用层软件设计上，采取了一个替代的办法。新浪播客提供了一个供播放器查询视频文件地址的接口。当用户打开视频播放页面的时候，播放器首先连接查询接口，通过接口获得视频文件所在的最优的镜像服务器地址，然后再到该服务器去下载视频文件。这样，用一次额外的查询获得了全部的控制性，而这次查询的通讯流量非常小，几乎可以忽略不计。CDN 中由域名解析获得的灵活性也保留了下来：由接口程序维护镜像网站列表及来访 IP 到镜像网站的对应表即可。镜像网站中不需要镜像所有的内容，而是只镜像更新速度较慢的视频文件。这是完全可以承受的。

2.4 网络层架构小结

从整个互联网络的高度来看网站架构，努力的方向是明确的：让用户就近取得内容，但又要在速度和可控制性之间作一个平衡。对于更新比较频繁内容，由于难以保持镜像网站之间的同步，则需要使用其他的辅助技术。

3 交换层架构

3.1 第四层交换简介

按照 OSI [16] 七层模型，第四层是传输层。传输层负责端到端通信，在 IP 协议栈中是 TCP 和 UDP 所在的协议层。TCP 和 UDP 数据包中包含端口号（port number），它们可以唯一区分每个数据包所属的协议和应用程序。接收端计算机的操作系统根据端口号确定所收到的 IP 包类型，并把它交给合适的高层程序。IP 地址和端口号的组合通常称作“插口(Socket)”。

第四层交换的一个简单定义是：它是一种传输功能，它决定传输不仅仅依据 MAC 地址(第二层网桥)或源/目标 IP 地址(第三层路由)，而且依据 IP 地址与 TCP/UDP (第四层) 应用端口号的组合(Socket) [17]。第四层交换功能就像是虚拟 IP，指向实际的服务器。它传输的数据支持多种协议，有 HTTP、FTP、NFS、Telnet 等。

以 HTTP 协议为例，在第四层交换中为每个服务器组设立一个虚拟 IP (Virtue IP, VIP)，每组服务器支持某一个或几个域名。在域名服务器 (DNS) 中存储服务器组的 VIP，而不是某一台服务器的真实地址。

当用户请求页面时，一个带有目标服务器组的 VIP 连接请求发送给第四层交换机。第四层交换机使用某种选择策略，在组中选取最优的服务器，将数据包中的目标 VIP 地址用实际服务器的 IP 地址取代，并将连接请求传给该服务器。第四层交换一般都实现了会话保持功能，即同一会话的所有包由第四层交换机进行映射后，在用户和同一服务器间进行传输[18]。

第四层交换按实现分类，分为硬件实现和软件实现。

3.2 硬件实现

第四层交换的硬件实现一般都由专业的硬件厂商作为商业解决方案提供。常见的有 Alteon [19]，F5 [20] 等。这些产品非常昂贵，但是能够提供非常优秀的性能和很灵活的管理能力。Yahoo 中国当初接近 2000 台服务器使用了三四台 Alteon 就搞定了 [21]。鉴于条件关系，这里不展开讨论。

3.3 软件实现

第四层交换也可以通过软件实现，不过性能比专业硬件稍差，但是满足一定量的压力还是可以达到，而且软件实现配置起来更灵活。软件四层交换常用的有 Linux 上的 LVS (Linux Virtual Server)，它提供了基于心跳 (heart beat) 的实时灾难应对解决方案，提高了系统的鲁棒性，同时提供了灵活的 VIP 配置和管理功能，可以同时满足多种应用需求 [22]。

4 服务器优化

4.1 服务器整体性能考虑

对于价值昂贵的服务器来说，怎样配置才能发挥它的最大功效，又不至于影响正常的服务，这是在设计网站架构的时候必须要考虑的。常见的影响服务器的处理速度的因素有：网络连接，硬盘读写，内存空间，CPU 速度。如果服务器的某一个部件满负荷运转仍然低于需要，而其他部件仍有能力剩余，我们将之称为性能瓶颈。服务器想要发挥最大的功效，关键的是消除瓶颈，让所有的部件都被充分的利用起来。

4.2 Socket 优化

以标准的 GNU/Linux 为例。GNU/Linux 发行版试图对各种部署情况都进行优化，这意味着对具体服务器的执行环境来说，标准的发行版可能并不是最优化的 [23]。GNU/Linux 提供了很多可调节的内核参数，可以使用这些参数为服务器进行动态配置，包括影响 Socket 性能的一些重要的选项。这些选项包含在 /proc 虚拟文件系统中。这个文件系统中的每个文件都表示一个或多个参数，它们可以通过 cat 工具进行读取，或使用 echo 命令进行修改。这里仅列出一些影响 TCP/IP 栈性能的可调节内核参数 [24]：

- `/proc/sys/net/ipv4/tcp_window_scaling` “1”（1 表示启用该选项，0 表示关闭，下同） 启用 RFC [25] 1323 [26] 定义的 `window scaling`；要支持超过 64KB 的窗口，必须启用该值。
- `/proc/sys/net/ipv4/tcp_sack` “1” 启用有选择的应答（Selective Acknowledgment），通过有选择地应答乱序接收到的报文来提高性能（这样可以让发送者只发送丢失的报文段）；对于广域网通信来说，这个选项应该启用，但是这也会增加对 CPU 的占用。
- `/proc/sys/net/ipv4/tcp_timestamps` “1” 以一种比重发超时更精确的方法（参阅 RFC 1323）来启用对 RTT 的计算；为了实现更好的性能应该启用这个选项。
- `/proc/sys/net/ipv4/tcp_mem` “24576 32768 49152” 确定 TCP 栈应该如何反映内存使用；每个值的单位都是内存页（通常是 4KB）。第一个值是内存使用的下限。第二个值是内存压力模式开始对缓冲区使用应用压力的上限。第三个值是内存上限。超过这个上限时可以将报文丢弃，从而减少对内存的使用。
- `/proc/sys/net/ipv4/tcp_wmem` “4096 16384 131072” 为自动调优定义每个 socket 使用的内存。第一个值是为 socket 的发送缓冲区分配的最少字节数。第二个值是默认值（该值会被 `wmem_default` 覆盖），缓冲区在系统负载不重的情况下可以增长到这个值。第三个值是发送缓冲区空间的最大字节数（该值会被 `wmem_max` 覆盖）。
- `/proc/sys/net/ipv4/tcp_westwood` “1” 启用发送者端的拥塞控制算法，它可以维护对吞吐量的评估，并试图对带宽的整体利用情况进行优化；对于 WAN 通信来说应该启用这个选项。

与其他调优努力一样，最好的方法实际上就是不断进行实验。具体应用程序的行为、处理器的速度以及可用内存的多少都会影响到这些参数对性能作用的效果。在某些情况中，一些认为有益的操作可能恰恰是有害的（反之亦然）。因此，需要逐一试验各个选项，然后检查每个选项的结果，最后得出最适合具体机器的一套参数。

如果重启了 GNU/Linux 系统，设置的内核参数都会恢复成默认值。为了将所设置的值作为这些参数的默认值，可以使用 `/etc/rc.local` 文件，在系统每次启动时自动将这些参数配置成所需要的值。

在检测每个选项的更改带来的效果的时候，GNU/Linux 上有一些非常强大的工具可以使用：

- `ping` 这是用于检查主机的可用性的最常用的工具，也可以用于计算网络带宽延时。
- `traceroute` 打印连接到特定网络主机所经过的一系列路由器和网关的路径（路由），从而确定每个 hop 之间的延时。
- `netstat` 确定有关网络子系统、协议和连接的各种统计信息。

- `tcpdump` 显示一个或多个连接的协议级的报文跟踪信息，其中包括时间信息，可以使用这些信息来研究不同协议的报文时间。
- `Ethereal` 以一个易于使用的图形化界面提供 `tcpdump`（报文跟踪）的信息，支持报文过滤功能。
- `iperf` 测量 TCP 和 UDP 的网络性能；测量最大带宽，并汇报延时和数据报的丢失情况。

4.3 硬盘级缓存

硬盘级别的缓存是指将需要动态生成的内容暂时缓存在硬盘上，在一个可接受的延迟时间范围内，同样的请求不再动态生成，以达到节约系统资源，提高网站承受能力的目的。Linux 环境下硬盘级缓存一般使用 `Squid` [27]。

`Squid` 是一个高性能的代理缓存服务器。和一般的代理缓存软件不同，`Squid` 用一个单独的、非模块化的、I/O 驱动的进程来处理所有的客户端请求。它接受来自客户端对目标对象的请求并适当地处理这些请求。比如说，用户通过浏览器想下载（即浏览）一个 web 页面，浏览器请求 `Squid` 为它取得这个页面。`Squid` 随之连接到页面所在的原始服务器并向服务器发出取得该页面的请求。取得页面后，`Squid` 再将页面返回给用户端浏览器，并且同时在 `Squid` 本地缓存目录里保存一份副本。当下一次有用户需要同一页面时，`Squid` 可以简单地从缓存中读取它的副本，直接返回给用户，而不用再次请求原始服务器。当前的 `Squid` 可以处理 HTTP，FTP，GOPHER，SSL 和 WAIS 等协议。

`Squid` 默认通过检测 HTTP 协议头的 `Expires` 和 `Cache-Control` 字段来决定缓存的时间。在实际应用中，可以显式的在服务器端脚本中输出 HTTP 头，也可以通过配置 `apache` 的 `mod_expires` 模块，让 `apache` 自动的给每一个网页加上过期时间。对于静态内容，如图片，视频文件，供下载的软件等，还可以针对文件类型（扩展名），用 `Squid` 的 `refresh_pattern` 来指定缓存时间。

`Squid` 运行的时候，默认会在硬盘上建两层 hash 目录，用来存储缓存的 `Object`。它还会在内存中建立一个 `Hash Table`，用来记录硬盘中 `Object` 分布的情况。如果 `Squid` 配置成为一个 `Squid` 集群中的一个的话，它还会建立一个 `Digest Table`（摘要表），用来存储其它 `Squid` 上的 `Object` 摘要。当用户端想要的资料本地硬盘上没有时，可以很快的知道应该去集群中的哪一台机器获得。在硬盘空间快要达到配置限额的时候，可以配置使用某种策略（默认使用 LRU: Least Recently Used-最近最少用）删除一些 `Object`，从而腾出空间 [28] [29]。

集群中的 `Squid Server` 之间可以有两种关系：第一种关系是：`Child` 和 `Parent`。当 `Child Squid Server` 没有资料时，会直接向 `Parent Squid Server` 要资料，然后一直等，直到 `Parent` 给它资料为止。第二种关系是：`Sibling` 和 `Sibling`。当 `Squid Server` 没有资料时，会先向 `Sibling` 的 `Squid Server` 要资料，如果 `Sibling` 没资料，就跳过它向 `Parent` 要或直接上原始网站去拿。

默认配置的 Squid，没有经过任何优化的时候，一般可以达到 50% 的命中率 [30]（图 4）。如果需要，还可以通过参数优化，拆分业务，优化文件系统等办法，使得 Squid 达到 90% 以上的缓存命中率。Squid 处理 TCP 连接消耗的服务器资源比真正的 HTTP 服务器要小的多，当 Squid 分担了大部分连接，网站的承压能力就大大增强了。

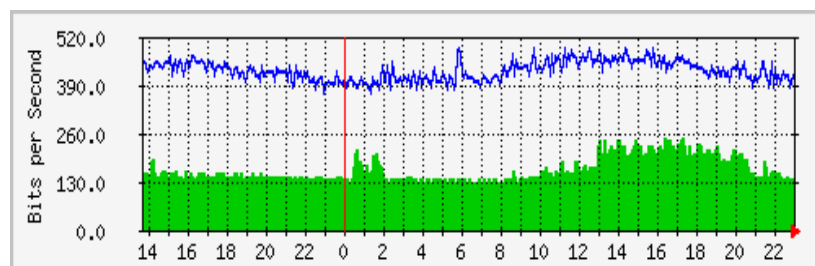


图
4 某网站使用
MRTG 工具检测
到的 Squid 命
中率
蓝线表示

Squid 的流量，绿色部分表示 Apache 流量

4.4 内存级缓存

内存级别的缓存是指将需要动态生成的内容暂时缓存在内存里，在一个可接受的延迟时间范围内，同样的请求不再动态生成，而是直接从内存中读取。Linux 环境下内存级缓存 Memcached [31] 是一个不错的选择。

Memcached 是 danga.com（运营 Live Journal [32] 的技术团队）开发的一套非常优秀的分布式内存对象缓存系统，用于在动态系统中减少数据库负载，提升性能。和 Squid 的前端缓存加速不同，它是通过基于内存的对象缓存来减少数据库查询的方式改善网站的性能，而其中最吸引人的一个特性就是支持分布式部署；也就是说可以在一群机器上建立一堆 Memcached 服务，每个服务可以根据具体服务器的硬件配置使用不同大小的内存块，这样，理论上可以建立一个无限大的基于内存的缓存系统。

Memcached 是以守护程序方式运行于一个或多个服务器中，随时接受客户端的连接操作，客户端可以由各种语言编写，目前已知的客户端 API 包括 Perl/PHP/Python/Ruby/Java/C#/C 等等[附录 1]。客户端首先与 Memcached 服务建立连接，然后存取对象。每个被存取的对象都有一个唯一的标识符 key，存取操作均通过这个 key 进行，保存的时候还可以设置有效期。保存在 Memcached 中的对象实际上是放置在内存中的，而不是在硬盘上。Memcached 进程运行之后，会预申请一块较大的内存空间，自己进行管理，用完之后再申请一块，而不是每次需要的时候去向操作系统申请。Memcached 将对象保存在一个巨大的 Hash 表中，它还使用 NewHash 算法来管理 Hash 表，从而获得进一步的性能提升。所以当分配给 Memcached 的内存足够大的时候，Memcached 的时间消耗基本上只是网络 Socket 连接了 [33]。

Memcached 也有它的不足。首先它的数据是保存在内存当中的，一旦服务进程重启（进程意外被关掉，机器重启等），数据会全部丢失。其次 Memcached 以 root 权限运行，而且 Memcached 本身没有任何权限管理和认证功能，安全性不足。第一条是 Memcached 作为内存缓存服务使用无法避免的，当然，如果内存中的数据需要保存，可以采取更改 Memcached

的源代码，增加定期写入硬盘的功能。对于第二条，我们可以将 Memcached 服务绑定在内网 IP 上，通过 Linux 防火墙进行防护。

4.5 CPU 与 IO 均衡

在一个网站提供的所有功能中，有的功能可能需要消耗大量的服务器端 IO 资源，像下载，视频播放等，而有的功能则可能需要消耗大量的服务器 CPU 资源，像视频格式转换，LOG 统计等。在一个服务器集群中，当我们发现某些机器上 CPU 和 IO 的利用率相差很大的时候，例如 CPU 负载很高而 IO 负责很低，我们可以考虑将该服务器上的某些耗 CPU 资源的进程换成耗 IO 的进程，以达到均衡的目的。均衡每一台机器的 CPU 和 IO 消耗，不仅可以获得更充分的服务器资源利用，而且还能够支持暂时的过载，遇到突发事件，访问流量剧增的时候，实现得体的性能下降 (Graceful performance degradation) [34]，而不是立即崩溃。

4.6 读写分离

如果网站的硬盘读写性能是整个网站性能提升的一个瓶颈的话，可以考虑将硬盘的读，写功能分开，分别进行优化。在专门用来写的硬盘上，我们可以在 Linux 下使用软件 RAID-0（磁盘冗余阵列 0 级）[35]。RAID-0 在获得硬盘 IO 提升的同时，也会增加整个文件系统的故障率——它等于 RAID 中所有驱动器的故障率之和。如果需要保持或提高硬盘的容错能力，就需要实现软件 RAID-1，4 或 5，它们能在某一个（甚至几个）磁盘驱动器故障之后仍然保持整个文件系统的正常运行 [36]，但文件读写效率不如 RAID-0。而专门用来读的硬盘，则不用如此麻烦，可以使用普通的服务器硬盘，以降低开销。

一般的文件系统，会综合考虑各种大小和格式的文件读，写效率，因而对特定的文件读或写的效率不是最优。如果有必要，可以通过选择文件系统，以及修改文件系统的配置参数来达到对特定文件的读或写的效率最大化。比如说，如果文件系统中需要存储大量的小文件，则可以使用 ReiserFS [37] 来替代 Linux 操作系统默认的 ext3 系统，因为 ReiserFS 是基于平衡树的文件系统结构，尤其对于大量文件的巨型文件系统，搜索速度要比使用局部的二分查找法的 ext3 快。ReiserFS 里的目录是完全动态分配的，因此不存在 ext3 中常见的无法回收巨型目录占用的磁盘空间的情况。ReiserFS 里小文件（< 4K）可以直接存储进树，小文件读取和写入的速度更快，树内节点是按字节对齐的，多个小文件可共享同一个硬盘块，节约大量空间。ext3 使用固定大小的块分配策略，也就是说，不到 4K 的小文件也要占据 4K 的空间，导致的空间浪费比较严重 [38]。但 ReiserFS 对很多 Linux 内核支持的不是很好，包括 2.4.3、2.4.9 甚至相对较新的 2.4.16，如果网站想要使用它，就必须安装与它配合的较好的 2.4.18 内核——一般管理员都不是很乐意使用太新的内核，因为在它上面运行的软件，都还没有经过大量的实践测试，也许有一些小的 bug 还没有被发现，但对于服务器来说，再小的 bug 也是不能接受的。ReiserFS 还是一个较为年轻的，发展迅速的文件系统，它相对于 ext3 来说有一个很大的缺陷就是，每次 ReiserFS 文件系统升级的时候，必须完全重新格式化整个磁盘分区。所以在选择使用的时候，需要权衡取舍 [39]。

5 应用程序层优化

5.1 网站服务器程序的选择

经统计 [40]，当前互联网上有超过 50% 的网站主机使用 Apache [41] 服务器程序。Apache 是开源界的首选 Web 服务器，因为它的强大和可靠，而且适用于绝大部分的应用场合。但是它的强大有时候却显得笨重，配置文件复杂得让人望而生畏，高并发情况下效率不太高。而轻量级的 Web 服务器 Lighttpd [42] 却是后起之秀，基于单进程多路复用技术，其静态文件的响应能力远高于 Apache。Lighttpd 对 PHP 的支持也很好，还可以通过 Fastcgi 方式支持其他的语言，比如 Python 等。虽然 Lighttpd 是轻量级的服务器，功能上不能跟 Apache 比，某些复杂应用无法胜任，但即使是大部分内容动态生成的网站，仍免不了会有一些静态元素，比如图片、JS 脚本、CSS 等等，可以考虑将 Lighttpd 放在 Squid 的前面，构成 Lighttpd->Squid->Apache 的一条处理链，Lighttpd 在最前面，专门处理静态内容的请求，把动态内容请求通过 Proxy 模块转发给 Squid，如果 Squid 中有该请求的内容且没有过期，则直接返回给 Lighttpd。新请求或者过期的页面请求交由 Apache 中的脚本程序来处理。经过 Lighttpd 和 Squid 的两级过滤，Apache 需要处理的请求大大减少，减少了 Web 应用程序的压力。同时这样的构架，便于把不同的处理分散到多台计算机上进行，由 Lighttpd 在前面统一分发。

在这种架构下，每一级都是可以单独优化的，比如 Lighttpd 可以采用异步 IO 方式，Squid 可以启用内存来缓存，Apache 可以启用 MPM (Multi-Processing Modules，多道处理模块) 等，并且每一级都可以使用多台机器来均衡负载，伸缩性好。

著名视频分享网站 YouTube 就是选择使用 Lighttpd 作为网站的前台服务器程序。

5.2 数据库选择

MySQL [43] 是一个快速的、多线程、多用户和健壮的 SQL 数据库服务器，支持关键任务、重负载系统的使用，是最受欢迎的开源数据库管理系统，是 Linux 下网站开发的首选。它由 MySQL AB 开发、发布和提供支持。

MySQL 数据库能为网站提供：

- 高性能。MySQL 支持海量，快速的数据库存储和读取。还可以通过使用 64 位处理器来获取额外的一些性能，因为 MySQL 在内部里很多时候都使用 64 位的整数处理。
- 易用性。MySQL 的核心是一个小而快速的数据库。它的快速连接，快速存取和安全可靠的特性使 MySQL 非常适合在互联网上使用。
- 开放性。MySQL 提供多种后台存储引擎的选择，如 MyISAM，Heap，InnoDB，Berkeley Db 等。缺省格式为 MyISAM。MyISAM 存储引擎与磁盘兼容的非常好 [44]。

- 支持企业级应用。MySQL 有一个用于记录数据改变的二进制日志。因为它是二进制的，这一日志能够快速地将数据的更改从一台机器复制（replication）到另一台机器上。即使服务器崩溃，这一二进制日志也能够保持完整。这一特性通常被用来搭建数据库集群，以支持更大的流量访问要求 [30]（图 5）。

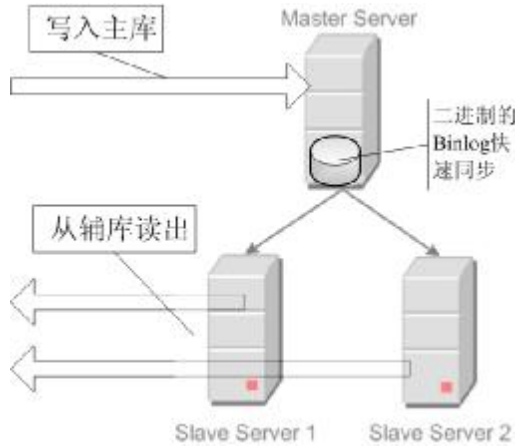


图
5 MySQL 主辅库模式集群示意

MySQL 也有一些它自身的缺陷，如缺乏图形界面，缺乏存储过程，还不支持触发器，参照完整性，子查询和数据表视图等，但这些功能都在开发者的 TO-DO 列表当中。这就是开源的力量：你永远可以期待更好。

国外的 Yahoo!，国内的新浪，搜狐等很多大型商业网站都使用 MySQL 作为后台数据库。对于一般的网站系统，无论从成本还是性能上考虑，MySQL 应该是最佳的选择。

5.3 服务器端脚本解析器的选择

目前最常见的服务器端脚本有三种：ASP(Active Server Pages)，JSP(Java Server Pages)，PHP (Hypertext Preprocessor) [45] [46]。

ASP 全名 Active Server Pages，以及它的升级 ASP.NET，是微软公司出品的一个 WEB 服务器端的开发环境，利用它可以产生和运行动态的、交互的、高性能的 WEB 服务应用程序。ASP 采用脚本语言 VBScript (C#) 作为自己的开发语言。但因为只能运行在 Windows 环境下，这里我们不讨论它。

PHP 是一种跨平台的服务器端的嵌入式脚本语言。它大量地借用 C，Java 和 Perl 语言的语法，并耦合 PHP 自己的特性，使 WEB 开发者能够快速写出动态生成页面。它支持目前绝大多数数据库。PHP 也是开源的，它的发行遵从 GPL 开源协议，你可以从 PHP 官方站点(<http://www.php.net>)自由下载到它的二进制安装文件及全部的源代码。如果在 Linux 平台上与 MySQL 搭配使用，PHP 是最佳的选择。

JSP 是 Sun 公司推出的新一代站点开发语言，是 Java 语言除 Java 应用程序和 Java Applet 之外的第三个应用。Jsp 可以在 Servlet 和 JavaBean 的支持下，完成功能强大的站点程序。作为采用 Java 技术家族的一部分，以及 Java 2（企业版体系结构）的一个组成部分，JSP 技术拥有 Java 技术带来的所有优点，包括优秀的跨平台性，高度可重用的组件设计，健壮性和安全性等，能够支持高度复杂的基于 Web 的应用。

除了这三种常见的脚本之外，在 Linux 下我们其实还有很多其他的选择：Python（Google 使用），Perl 等，如果作为 CGI 调用，那么可选择范围就更广了。使用这些不太常见的脚本语言的好处是，它们对于某些特殊的应用有别的脚本所不具有的优势；不好的地方是，这些脚本语言在国内使用的人比较少，当碰到技术上的问题的时候，能找到的资料也较少。

5.4 可配置性

在大型网站开发过程中，不管使用什么技术，网站的可配置性是必须的。在网站的后期运营过程中，肯定会有很多的需求变更。如果每一次的需求变更都会导致修改源代码，那么，这个网站的开发可以说是失败的。

首先，也是最重要的一点，功能和展示必须分开。PHP 和 JSP 都支持模板技术，如 PHP 的 Smarty, Phplib, JSP 的 JSTL (JSP Standard Tag Library) 等。核心功能使用脚本语言编写，前台展示使用带特殊标签的 HTML，不仅加快了开发速度，而且方便以后的维护和升级 [47]。

其次，对于前台模板，一般还需要将页面的头，尾单独提取出来，页面的主体部分也按模块或者功能拆分。对 CSS, JS 等辅助性的代码，也建议以单独的文件形式存放。这样不仅方便管理，修改，而且还可以在用户访问的时候进行缓存，减少网络流量，减轻服务器压力。

再次，对于核心功能脚本，必须将与服务器相关的配置内容，如数据库连接配置，脚本头文件路径等，与代码分离开。尤其当网站使用集群技术，CDN 加速等技术的时候，每一台服务器上的配置可能都会不一样。如果不使用配置文件，则需要同时维护几份不同的代码，很容易出错。

最后，应该尽量做到修改配置文件后能实时生效，避免修改配置文件之后需要重启服务程序的情况。

5.5 封装和中间层思想

在功能块层次，如果使用 JSP，基于纯面向对象语言 Java 的面向对象思想，类似数据库连接，会话管理等基本功能都已经封装成类了。如果使用 PHP，则需要在脚本代码中显式的封装，将每一个功能块封装成一个函数，一个文件或者一个类。

在更高的层次，可以将网站分为表示层，逻辑层，持久层，分别进行封装，做到当某一层架构发生变化时，不会影响到其他层。比如新浪播客在一次升级的时候，将持久层的数据库由原来的集中式改为分布式架构，因为封装了数据库连接及所有操作 [附录 2]，做到了不修改任何上层代码，平稳的实现了过渡。近来流行的 MVC 架构，将整个网站拆分成 Model（模型/逻辑）、View（视图/界面）、Controller（控制/流程）三个部分，而且有很多优秀的代码框架可供选择使用，像 JSP 的 Struts, Spring, PHP 的 php. MVC, Studs 等。使用现成的代码框架，可以使网站开发事半功倍。

6 扩容、容错处理

6.1 扩容

一个大型网站，在设计架构的时候，必须考虑到以后可能的容量扩充。新浪播客在设计时充分地考虑了这一点。对于视频分享类网站来说，视频存储空间消耗是巨大的。新浪播客在主存储服务器上，采用配置文件形式指定每一个存储盘柜上存储的视频文件的 ID 范围。当前台服务器需要读取一个视频的时候，首先通过询问主存储服务器上的接口获得该视频所在的盘柜及目录地址，然后再去该盘柜读取实际的视频文件。这样如果需要增加存储用的盘柜，只需要修改配置文件即可，前台程序丝毫不受影响。

新浪播客采用 MySQL 数据库集群，在逻辑层封装了所有的数据库连接及操作。当数据库存储架构发生改变的时候，如增加一台主库，将某些数据表独立成库，增加读取数据用的从库等，都只需要修改封装了的数据库操作类，上层代码不用修改。

新浪播客的前台页面服务器使用 F5 公司的硬件第四层交换机，网通，电信分别导向不同的虚拟 IP，每一个虚拟 IP 后面又有多个服务器提供服务。当访问流量增大的时候，可以很方便往虚拟 IP 后面增加服务器，分担压力。

6.2 容错

对于商业性网站来说，可用性是非常重要的。7*24 的访问要求网站具有很强的容错能力。错误包括网络错误，服务器错误以及应用程序错误。

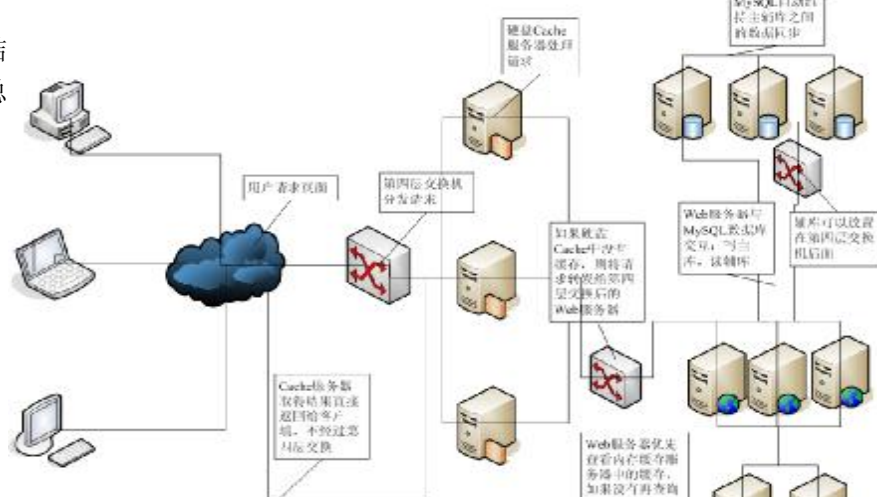
2006 年 12 月 27 日台湾东部外海发生里氏 7.6 级地震，造成途径台湾海峡的多条海底电缆中断，导致许多国外网站，像 MSN，NBA，Yahoo！（英文主站）等国内无法访问，但也有例外，以 Google 为代表的在国内建设有分布式数据节点的很多网站却仍然可以访问。虽然说地震造成断网是不可抗原因，但如果在这种情况下网站仍然可以访问，无疑能给网站用户留下深刻的印象。这件事情给大型商业网站留下的教训是：网站需要在用户主要分布区域保持数据存在，以防止可能的网络故障。

对于服务器错误，一般采取冗余设计的方法来避免。对于存储服务器（主要是负责写入的服务器），可以使用 RAID（冗余磁盘阵列）；对于数据库（主要是负责写入的主库），可以采用双主库设计 [30]；对于提供服务的前台，则可以使用第四层交换的集群，由多台服务器同时提供服务，不仅分担了流量压力，同时还可以互相作为备份。

在应用层程序中，也要考虑“用户友好”的出错设计。典型例子如 HTTP 404 出错页面，程序内部错误处理，错误返回提示等，尽可能的做到人性化。

7 总结

7.1 总



及展望
总结

对于一个高并发高流量的网站来说，任何一个

环节的瓶颈都会造成网站性能的下降，影响用户体验，进而造成巨大的经济损失。在全互联网层面，应该使用分布式设计，缩短网站与用户的网络距离，减少主干网上的流量，以及防止在网络意外情况下网站无法访问的问题。在局域网层面，应该使用服务器集群，一方面可以支撑更大的访问量，另一方面也作为冗余备份，防止服务器故障导致的网站无法访问。在单服务器层面，应该配置操作系统，文件系统及应用层软件，均衡各种资源的消耗，消除系统性能瓶颈，充分发挥服务器的潜能。在应用层，可以通过各种缓存来提升程序的效率，减少服务器资源消耗（图6）。另外，还需要合理设计应用层程序，为以后的需求变更，扩容做好准备。

图6 典型高并发高流量网站的架构

在每一个层次，都需要考虑容错的问题，严格消除单点故障，做到无论应用层程序错误，服务器软件错误，服务器硬件错误，还是网络错误，都不影响网站服务。

7.2 展望

当前 Linux 环境下有著名的 LAMP（Linux+Apache+MySQL+PHP/PERL/PYTHON）网站建设方案，但只是针对一般的中小网站而言。对于高并发高流量的大型商业网站，还没有一个完整的，性价比高的解决方案。除去服务器，硬盘，带宽等硬件投资外，还需要花费大量的预算和时间精力在软件解决方案上。

随着互联网的持续发展，Web2.0 的兴起，在可以预见的未来里，互联网的用户持续增多，提供用户参与的网站不断增加，用户参与的内容日益增长，越来越多的网站的并发量，访问量会达到一个新的高度，这就会促使越来越多的个人，公司以及研究机构来关注高并发高流量的网站架构问题。就像 Web1.0 成就了无数中小网站，成就了 LAMP 一样，Web2.0 注定也会成就一个新的，高效的，成本较低的解决方案。这个方案应该包括透明的第三方 CDN 网络加速服务，价格低廉的第四层甚至更高层网络交换设备，优化了网络性能的操作系统，优化了读写性能，分布式，高可靠的文件系统，揉合了内存，硬盘等各个级别缓存的 HTTP 服务器，更为高效的服务器端脚本解析器，以及封装了大部分细节的应用层设计框架。

技术的进步永无止境。我们期待互联网更为美好的明天。

参考文献

[1] Robert Hobbes' Zakon, Hobbes' Internet Timeline v8.2 , available at <http://www.zakon.org/robert/internet/timeline/>

[2] Global Reach Inc., Global Internet Statistics (by language), available at

<http://www.glreach.com/globalstats/index.php3>

[3] 中国互联网络信息中心, 第十九次中国互联网络发展状况统计报告, available at: <http://www.cnnic.net.cn/index/0E/index.htm>

[4] Web2.0, Definition available at <http://www.wikilib.com/wiki/Web2.0>

[5] Alexa Internet, Inc. <http://www.alexa.com/>

[6] Yahoo! Inc. <http://www.yahoo.com/>

[7] eBay Inc. 著名的网上拍卖网站, <http://www.ebay.com/>

[8] Chet Dembeck, Yahoo! Cashes In On eBay's Outage, available at: <http://www.ecommercetimes.com/perl/story/545.html>

[9] YouTube, Inc. <http://www.youtube.com/>

[10] 数据来源: 互联网周刊, 2007 年第 3 期

[11] 新浪网技术(中国)有限公司, <http://www.sina.com.cn/>

[12] 数据来源: 新浪播客改版公告, available at: <http://games.sina.com.cn/x/n/2007-04-16/1427194553.shtml>

[13] 邓宏炎, 叶娟丽, 网络参考文献初探, 武汉大学学报: 人文社会科学版, 2000

[14] 彭湘凯, CDN 网络及其应用, 微计算机信息, 2005 年 02 期

[15] 数据来源: ChinaCache, <http://www.chinacache.com/>

[16] Open System Interconnect, 开放式系统互联模型, 1984 年由国际标准化组织(ISO)提出的一个开放式网络互联参考模型, 参考 <http://www.iso.org/>

[17] 凌仲权, 丁振国, 基于第四层交换技术的负载均衡, 中国数据通信, 2003

[18] 陈明锐, 邱钊, 黄曦, 黄俊, 智能负载均衡技术在高负荷网站上的应用, 广西师范大学学报(自然科学版), 2006 年 04 期

[19] Alteon Inc. <http://www.alteon.com/>

[20] F5 Networks, Inc. <http://www.f5.com.cn/>

[21] 数据来源: <http://www.toptee.com/blog/archives/71.html>

[22] 傅明, 程晓恒, 王玮, 基于 Linux 的服务器负载均衡性访问的解决方案, 计算机系统应用, 2001 年 09 期

[23] Ming-Wei Wu, Ying-Dar Lin, Open source software development: an overview, Computer, 2001 - ieeexplore.ieee.org

[24] 王海花, 杨斌, Linux TCP/IP 协议栈的设计及实现特点, 云南民族大学学报(自然科学版), 2007 年 01 期

[25] Requests for Comments (RFC), the publication vehicle for technical specifications and policy documents produced by the IETF (Internet Engineering Task Force), the IAB (Internet Architecture Board), or the IRTF (Internet Research Task Force), <http://www.ietf.org/rfc.html>

[26] RFC 1323, <http://www.ietf.org/rfc/rfc1323.txt?number=1323>

[27] Squid web proxy cache team, <http://www.squid-cache.org/>

[28] 马俊昌, 古志民, 网络代理缓存 Squid 存储系统分析, 计算机应用, 2003 年 10 期

[29] 韩向春, 郭婷婷, 林星宇, 丰保杰, 集群缓存系统中代理缓存技术的研究, 计算机工程与设计, 2006 年 20 期

[30] Brad Fitzpatrick, LiveJournal's Backend, A history of scaling, oscon 2005, <http://www.danga.com/words/>

[31] Danga Interactive, <http://www.danga.com/memcached/>

[32] LiveJournal, 著名的博客托管商 (BSP), <http://www.livejournal.com/>

[33] Brad Fitzpatrick, Distributed caching with memcached, Linux Journal, Volume 2004, Issue 124, Page 5, August 2004

[34] 周枫, 面向 Internet 服务的可扩展集群对象存储及磁盘日志缓存技术研究, 清华大学硕士毕业论文, 2002

[35] 陈赞, 杨根科, 吴智铭, RAID 系统中 RAID 级别的具体实现算法, 微型电脑应用, 2003 年 06 期

[36] 陈平仲, 硬件实现 RAID 与软件实现 RAID 的比较, 现代计算机 (专业版), 2005 年 01 期

[37] NAMESYS, <http://www.namesys.com/>

[38] D Bobbins, Advanced file system implementor's guide: Journaling and ReiserFS, IBM's Developer Works Journal, June, 2001

[39] 刘章仪, Linux 文件系统分析, 贵州工业大学学报 (自然科学版), 2002 年 04 期

[40] 数据来源:
http://news.netcraft.com/archives/2007/04/02/april_2007_web_server_survey.html

[41] The Apache Software Foundation, <http://httpd.apache.org/>

[42] Lighttpd, <http://www.lighttpd.net/>

[43] MySQL AB, <http://www.mysql.com/>

[44] 顾治华, 忽朝俭, MySQL 存储引擎与数据库性能, 计算机时代, 2006 年 10 期

[45] The PHP Group, <http://www.php.net/>

[46] 范云芝, 动态网页制作技术 ASP、PHP 和 JSP 比较分析, 电脑知识与技术 (学术交流), 2005 年 10 期

[47] 王耀希, 王丽清, 徐永跃, 利用模板技术实现 B/S 研发过程的分离与并行, 计算机应用研究, 2004

附 录

[附录 1]

1. Memcache 的客户端 PHP 封装

```
class memcache_class
{
function memcache_class()
{
}

/**
 * 用 post 方法, 执行 memcache 的写入操作
 * $data 参数, 允许是 php 的数组。
 * exp 参数是设定的超时时间, 单位是秒。
 */
function p_memcache_write($key, $data, $exp=3600)
{
    $mmPageStartTime = microtime();
    $ip = MEMCACHE_SERVER_IP;
    $port = MEMCACHE_SERVER_PORT;
    $type = MEMCACHE_SERVER_TYPE;

    //对$data 进行序列化, 允许$data 是数组
    $data = serialize($data);

    //对$data 进行压缩
    //$data = gzcompress ($data);
```

```
$submit=array( type => $type,
cmd => "set",
key => $key,
data => $data,
exp => $exp
);

$ret = memcache_class::posttohost($query, $submit);
return $ret;
}
```

```
/**
```

```
* 用 post 方法，执行 memcache 的读出操作
```

```
*/
```

```
function p_memcache_read($key)
```

```
{
```

```
$mmPageStartTime = microtime();
```

```
$ip = MEMCACHE_SERVER_IP;
```

```
$port = MEMCACHE_SERVER_PORT;
```

```
$type = MEMCACHE_SERVER_TYPE;
```

```
$submit=array( type => $type,
```

```
cmd => "get",
```

```
key => $key
```

```
);
```

```
$res = memcache_class::posttohost($query, $submit);
```

```
//对$res 进行解压缩
```

```
//$res = gzuncompress($res);
```

```
//对$res 进行反序列化，允许$res 是数组
```

```

$res = unserialize($res);

return $res;

}

/**
 * 执行 post 的函数
 */

function posttohost($url, $data)
{
    $mmPageStartTime = microtime();

    $url = parse_url($url);

    $encoded = "";

    while (list($k,$v) = each($data))
    {
        $encoded .= ($encoded ? "&" : "");

        $encoded .= rawurlencode($k). "=". rawurlencode($v);
    }

    for ($i = 0; $i < 3; $i++)
    {
        $fp = @fsockopen($url['host'], $url['port'], $errno, $errstr, 1);

        if ($fp)
            break;
    }

    if (!$fp)
    {
        return "";
    }

    @stream_set_timeout($fp, 2);

    @fputs($fp, sprintf("POST %s%s HTTP/1.0\n", $url['path'], $url['query'] ?
    "?" : "", $url['query']));

    @fputs($fp, "Host: $url[host]\n");

```

```

@fputs($fp, "Content-type: application/x-www-form-urlencoded\n");
@fputs($fp, "Content-length: " . strlen($encoded) . "\n");
@fputs($fp, "Connection: close\n\n");
@fputs($fp, "$encoded\n");
$line = @fgets($fp, 1024);
if (!eregi("^HTTP/1\\. 200", $line)) return;
$results = "";
$header = 1;
while(!feof($fp))
{
$line = @fgets($fp, 1024);
if ($header && ($line == "\n" || $line == "\r\n"))
{
$header = 0;
}
elseif (!$header)
{
$results .= $line;
}
}
@fclose($fp);
return $results;
}
}

```

2. 使用示例

```

$out="";

if (MEMCACHE_FLAG === true)
{
$memcache_key = md5(trim($key));

```

```

$time_before = getmicrotime();

$mdata = memcache_class::p_memcache_read($memcache_key);

$time_after = getmicrotime();

$memcache_read_time = $time_after - $time_before;

if (strlen($mdata) >= MIN_RESULT) {

    $out = $mdata;

    $memhit = 1;

    memcached_log("CACHE_HIT");

}

else {

    $memhit = 0;

    memcached_log("CACHE_NOT_HIT");

}

if (!(strlen($out) >= MIN_RESULT))

{

    $query = get_query();

    $time_before=getmicrotime();

    $out = http_read($MySQLHost,$MySQLPort,$query,&$errstr,10);

    $time_after=getmicrotime();

}

$len = strlen($out);

if(MEMCACHE === true && $memhit <= 0)

{

    $memcache_key = md5(trim($key));

    $time_before = getmicrotime();

    memcache_class::p_memcache_write($memcache_key, $out, MEMCACHE_TIME);

    $time_after = getmicrotime();

    $memcache_write_time = $time_after - $time_before;

```

```
memcached_log("CACHE_WRITE");  
}
```

[附录 2]

MySQL wrap class

```
<?php
```

```
class mysqlRpc
```

```
{
```

```
var $_hostWrite = '';
```

```
var $_userWrite = '';
```

```
var $_passWrite = '';
```

```
var $_hostRead = '';
```

```
var $_userRead = '';
```

```
var $_passRead = '';
```

```
var $_dataBase = '';
```

```
var $db_write_handle = null;
```

```
var $db_read_handle = null;
```

```
var $db_last_handle = null;
```

```
var $_cacheData = array();
```

```
var $mtime = 60;
```

```
function mysqlRpc($database, $w_servername, $w_username, $w_password,  
$r_servername='', $r_username='', $r_password='') {}
```

```
function connect_write_db() {}
```

```
function connect_read_db() {}
```

```
function query_write($sql, $return = false) {}
```

```
function query_read($sql, $return = false) {}
```

```
function query_first($sql, $return = false) {}
```

```
function insert_id() {}
```



```
function affected_rows(){}

function escape_string($string){}

function fetch_array($queryresult, $type = MYSQL_ASSOC){}

}
```

作为亚洲最大的交易型电子商务网站，淘宝的每一项数字都是惊人的：2007 年上半年，淘宝总成交额突破 157 亿元人民币，接近其 2006 年 169 亿元的全年成交额，相当于 122 个家乐福或 150 个沃尔玛大卖场。和去年同期相比，淘宝成交额增长了近 200%。目前，淘宝的注册用户超过了 4500 万，商品数 9000 多万，而全球最大的 C2C 网站 eBay 的商品数是 1.1 亿左右。这些现实中的数字映射到网络中，便是高速膨胀的海量数据。路鹏介绍说，在淘宝的数据管理架构中，首先做的工作便是对信息进行分类管理，比如商品的类目、属性等，淘宝采用了一些分库策略，将其分放在不同的数据库。另外，根据数据的重要性，淘宝又将数据分为核心数据和非核心数据。其中，核心数据包括用户信息和交易信息等数据，而非核心数据包括商品描述、图片、商品评价、论坛等信息。这两类数据仍然采用了不同的存储方法，分放在不同的数据库里，其迁移策略和负载的均衡策略不一样。淘宝每天都做一次检查，将其 3~6 个月内不活跃的数据从在线迁移到近线的存储数据库里。淘宝每天的页面浏览量是 2 亿多，如何在高并发量的状态下加快访问速度？这其实也和数据的分类和存储有关。“从 IT 架构去规划的话，淘宝的数据分为静态和动态，每天的页面浏览量里，有将近 70% 是静态数据。”路鹏说。那些网页上能看到的图片、商品描述等信息，淘宝将之归为静态数据，采用了 CDN(Content Delivery Network，内容分发网络)技术，静态数据除了存放于杭州的两个数据中心外，淘宝还在上海、天津、杭州、宁波等城市建立了 CDN 分发点，旨在通过将网站的内容发布到最接近用户的缓存服务器内，使用户就近访问缓存取得需要的内容，提高网站的响应速度。而动态数据则是用户在进行注册、交易、评价等行为时产生的数据，淘宝通过优化后台数据库和应用层系统，把动态的数据快速呈现给用户。对于每天不断涌现然后逐渐沉寂的数据，淘宝在它们生命周期的各个阶段也采用了不同的存储管理方案。“它们分为 3 大类，在线、近线和离线。”路鹏说。其中，在线是指那些正在出售中的商品数据，仅仅是在线的数据就超过 10TB，淘宝采用了 FC(Fiber Connector)光纤存储技术，尽管成本高，但是可以加快访问速度；近线包括了已经下架的商品数据、交易评价等数据，用户还有可能访问到，淘宝采用了以串行方式传输数据 SATA(Serial ATA，串行 ATA)技术，这样既能保证用户的正常访问，相比于光纤存储技术，更加经济；离线则是包括了已经交易完成的数据以及相关的信息，淘宝将其作为历史数据，用卡库、光盘等介质进行存储。

I 网站架构的高性能和可扩展性

2007-10-09 – 4:33 下午

高性能和可扩展性是网站架构中非常重要的问题，尤其对于 Web2.0 站点来说，要应付高并发的访问，必须充分考虑到这些问题。

什么是高性能和可扩展性？高性能通俗地说访问速度要快。服务器对一个页面请求的响应时间通常必须控制在 10s 以内，否则就会产生很糟糕的用户体验。前一段时间我们对[育儿网](#)的性能做了一些提升：使用 icegrid 解决 ice 请求堆积的问题、加入缓存(memcached)、优化一些数据库的设计，网站的访问量也发生的很明显的增长。缓存、代码优化等都是提高性能的常用手段。

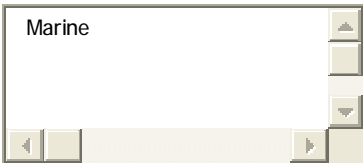
可扩展性就是当访问量增加时，为了维持高性能所要付出的成本。一台服务器的处理能力是有限的，因此访问量进一步增加时我们不得不加入更多的服务器，这时一个好的架构就尤其重要，通常 web server、数据库、中间层等都要考虑到可扩展性，如数据层可以用 Mysql 的 Master-slave 结构，前端可以用 dns 轮询，squid 等实现负载均衡等。在[育儿网](#)我们用 icegrid 实现了 ice 层的可扩展性，现在需要增加 ice server 时只需增加服务器和修改 registry 的配置就可以了，而不需改变客户端的代码。

总而言之，要用最低的成本获得最高的性能！

新浪这样的大型网站首页如何架构 **【已结贴】**

发表于： 2007-06-05 23:51:50 楼主

新浪、搜狐、淘宝等这样的大型网站，首页的架构设计怎样比较合理？

- [marine_chen](#)
 据我所知，新浪、搜狐这样的新闻媒体为主的，数据实时性要求不高，可以生成静态页实现。淘宝的数据实时性要求较高，也生成静态页来实现？还是用一些 cache 缓存来实现？
欢迎大家探讨
- 等级：

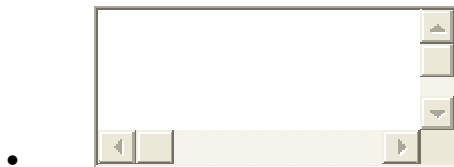
问题点数： 100 回复次数： 32 [显示所有回复](#)

- [Rachael1001](#)
 发表于： 2007-06-06 00:35:41 楼 得分:0
估计甚麽架构都渗杂一些
- 等级：
- [zhj92lxs](#)
 发表于： 2007-06-06 01:30:56 楼 得分:0
都用一下把
- 等级：

发表于： 2007-06-06 01:48:59 楼 得分:0

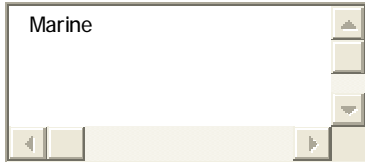
还没那水平 在这里等待高手

- [jsczxy2](#)



等级:

marine_chen



发表于: 2007-06-06 08:51:074 楼 得分:0

我们以前是用过 **cache**, 效果还行

等级:

发表于: 2007-06-06 10:06:435 楼 得分:0

楼主, 我要真城地告诉你的是, 新浪这样的大网站, 不光是说做一个首页就可以好的。因为是千万人同时访问的网站, 所以一般是有多个数据库同时工作的, 说明白一点就是数据库集群和并发控制。另外还有一点的是, 那些网站的静态化网页并不是真的, 而是通过动态网页与静态网页网址交换出现的假象, 这可以用 **urlrewrite** 这样的开源网址映射器实现。这样的网站实时性也是相对的, 因为在数据库复制数据的时候有一个过程, 一般在技术上可以用到 **hibernate** 和 **ecache**, 但是如果要使网站工作地更好, 可以使用 **EJB** 和 **websphere**, **weblogic** 这样大型的服务器来支持, 并且要用 **oracle** 这样的大型数据库。

cucuchen

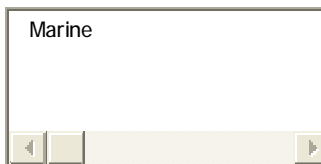


等级:

发表于: 2007-06-06 11:08:586 楼 得分:0

楼主, 我要真城地告诉你的是, 新浪这样的大网站, 不光是说做一个首页就可以好的。因为是千万人同时访问的网站, 所以一般是有多个数据库同时工作的, 说明白一点就是数据库集群和并发控制。另外还有一点的是, 那些网站的静态化网页并不是真的, 而是通过动态网页与静态网页网址交换出现的假象, 这可以用 **urlrewrite** 这样的开源网址映射器实现。这样的网站实时性也是相对的, 因为在数据库复制数据的时候有一个过程, 一般在技术上可以用到 **hibernate** 和 **ecache**, 但是如果要使网站工作地更好, 可以使用 **EJB** 和 **websphere**, **weblogic** 这样大型的服务器来支持, 并且要用 **oracle** 这样的大型数据库。

marine_chen



等级:

感谢回复。

数据库集群、并发控制、**weblogic**、**oralce** 等, 这些都是硬件方面的, 我也都运用过, 效果还行。

我想知道, 在具体技术细节的运用上有没有什么可以借鉴的,

hibernate 的 ehcache 我也用过, 本身也是有一些缺点不够完善, 而且从我的经验来看, hibernate 不太适合大型系统的运用, 从这点来看还不如 ibatis。

cache 方面, 像 swarmcache、memcache 都只是一些缓存的概念, 用哪个都各有利弊, 想知道有没有从技术角度出发能提高性能的方法呢

发表于: 2007-06-06 11:33:227 楼 得分:20

楼主, 我在 made-in-china.com 做过设计, 通过我的经验, 我认为一个网站要做过效率高, 不过是一个程序员的事情。在性能优化上要数据库和程序齐头并进! 缓存也是两方面同时入手。第一: 数据库缓存和数据库优化, 这个由 dba 完成(而且这个有非常大的潜力可挖, 只是由于我们都是程序员而忽略了他而已)。第二: 程序上的优化, 这个非常的有讲究, 比如说重要一点就是要规范 S Q L 语句, 少用 in 多用 or, 多用 preparestatement, 另外避免程序冗余如查找数据少用双重循环等。另外选用优秀的开源框架加以支持, 我个人认为中后台的支持是最最重要的, 可以选取 spring+ibatis。因为 ibatis 直接操作 SQL 并有缓存机制。spring 的好处就不用我多说了, I O C 的机制可以避免 new 对象, 这样也节省开销! 具我分析, 绝大部分的开销就是在 NEW 的时候和连接数据库时候产生的, 请你尽量避免。另外可以用一些内存测试工具来做一个 demo 说明 hibernate 和 ibatis 谁更快! 前台你想用什么就用什么, struts,webwork 都成, 如果觉得自己挺牛 X 可以试试用 tapestry。

- [cucuchen](#)



- 等级:

发表于: 2007-06-06 11:35:108 楼 得分:0

更正: 我认为一个网站要做过效率高, 不过是一个程序员的事情——》我认为一个网站要做的效率高, 不光是一个程序员的事情。

- [cucuchen](#)

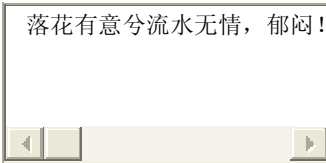


- 等级:

发表于: 2007-06-06 11:43:219 楼 得分:0

我同意 marine_chen(覆雨翻云) 的观点, 后缀名为 htm 或者 html 并不能说明程序生成了静态页面, 可能是通过 url 重写来实现的, 为的只不过是搜索引擎中提升自己网站的覆盖面积罢了。

- [didibaba](#)



- 等级:

其实用数据库也未必不能解决访问量巨大所带来的问题, 作成静态文件硬盘的寻址时间也未必少于数据库的搜索时间, 当然对资料的索引要下一翻工夫。

我自己觉得门户往往也就是当天、热门的资料点击率较高，将其做缓存最多也不过 1~2G 的数据量吧，别说服务器，个人电脑，1~2G 小意思。

拿网易新闻来说

<http://news.163.com/07/0606/09/3GA0D10N00011229.html>

格式化一下，方便理解：**http://域名/年/月/日/新闻所属分类/新闻 ID.html**

我们可以把当天发布的、热门的、浏览量大的作个缓存，用 **hashtable** (**key**: 年-月-日-分类-**ID**, **value**: 新闻对象)，静态将其放到内存（速度绝对快过硬盘寻址静态页面）。

这样可以大大增加一台计算机的处理速度。至于一台机器不够处理的，那是 **httpserver** 集群来路由的问题了。

生成静态页面其实是比较笨的做法啊：

- 1、增加了程序的复杂度
- 2、不利于管理资料
- 3、速度也不是最快
- 4、伤硬盘，哈哈

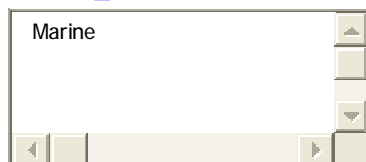
发表于：2007-06-06 11:45:50 10 楼 得分:0

你说的都很有道理，也给我很大启发，非常感谢。

我现在正在设计一个电子商务网站，像首页、二级页面之类的还没考虑好用什么样的方式实现，我发这个帖子希望能在这方面有所启发。

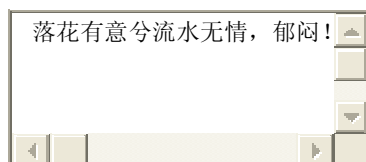
数据库优化、程序优化这个是必须的，框架方面我也选了 **webwork+ibatis+spring, tapestry** 了解一些不过还没具体实践，难道 **tapestry** 的性能更好？

• [marine_chen](#)



• 等级:

• [didibaba](#)

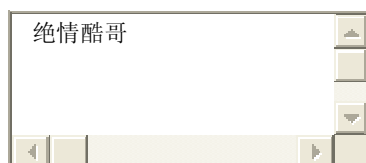


• 等级:

发表于：2007-06-06 11:47:25 11 楼 得分:0

其他旧的资料流量不大，用一般的处理方法能应付~

• [cucuchen](#)

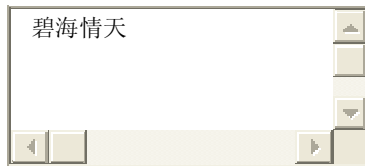


发表于：2007-06-06 11:53:02 12 楼 得分:0

回楼主，你的方案和我的方案是不谋而合的。我也是：**webwork+ibatis+spring, tapestry** 的确很优秀，但是难度大，但是他的好处是程序和美术分离，而且是事件机制，也非常棒，考虑到学习曲线，你用 **ww** 也不错呀!!!

- 等级:

- [theforever](#)



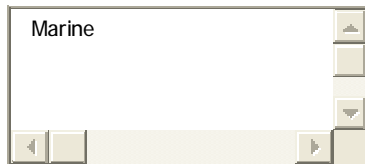
发表于: 2007-06-06 12:36:56 13 楼 得分:0

空说无益,检验真理的标准唯有实践.

写个数据生成程序,在块大硬盘上实际来来,用结果说话.

- 等级:

- [marine_chen](#)



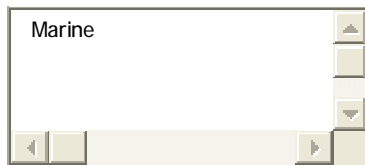
发表于: 2007-06-06 12:39:12 14 楼 得分:0

didibaba 说的也有道理。

其实从我的角度是不想做静态页的,还得单独增加服务器用来存储,复杂而且增加开销。

- 等级:

- [marine_chen](#)



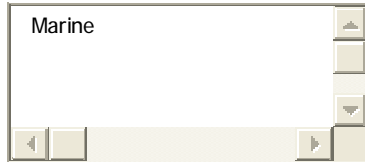
发表于: 2007-06-06 12:42:21 15 楼 得分:0

网站首页初始化一般都是如何实现?

难道每次打开都拉一次数据? 比如各个小模块都怎么关联?

- 等级:

- [marine_chen](#)



发表于: 2007-06-06 12:42:46 16 楼 得分:0

theforever(碧海情天) 的意思是用静态页?

- 等级:

发表于: 2007-06-06 12:54:46 17 楼 得分:0

- [tiandiqing](#)



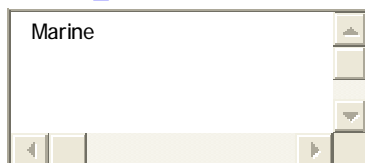
新浪的后台发布我接触过

是 perl php mysql 的,有几个人写的底层东西,然后频道的开发人员在上面进行二次开发。

那个系统很灵活,都是生成静态页面的

- 等级:

- [marine_chen](#)



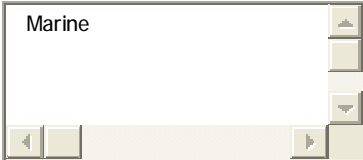
发表于: 2007-06-06 12:57:22 18 楼 得分:0

cucuchen(绝情酷哥) ,这个框架网站首页初始化怎样比较好?

- 等级:

- [marine_chen](#)


发表于: 2007-06-06 12:57:49 19 楼 得分:0

-  新浪的后台发布我接触过
- 是 **perl php mysql** 的，有几个人写的底层东西，然后频道的开发人员在上面进行二次开发。

- 等级: 那个系统很灵活，都是生成静态页面的

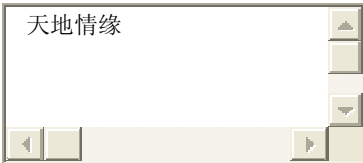
静态页的速度应该比动态的要快一些

发表于: 2007-06-06 12:59:1620 楼 得分:0

- [tiandiqing](#)  每签发一条新闻，就会生成静态页面，然后发往前端的 **web** 服务器，前端的 **web** 都是做负载均衡的。另外还有定时的程序，每 **5-15** 分钟自动生成一次。做一个大的网站远没有想象中那么简单。服务器基本就要百十个的

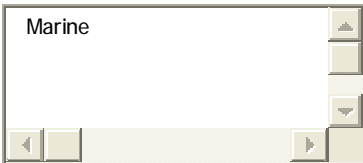
- 等级:

- [tiandiqing](#) 发表于: 2007-06-06 13:00:3121 楼 得分:0

-  如果哪位想要做大型的门户网站系统，可以找我联系，我可以出技术解决方案，呵呵

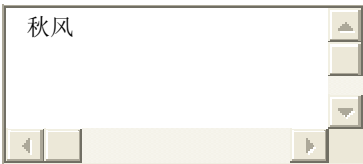
- 等级:

发表于: 2007-06-06 13:16:3322 楼 得分:0

- [marine_chen](#)  每签发一条新闻，就会生成静态页面，然后发往前端的 **web** 服务器，前端的 **web** 都是做负载均衡的。另外还有定时的程序，每 **5-15** 分钟自动生成一次。做一个大的网站远没有想象中那么简单。服务器基本就要百十个的

- 等级: 负载均衡、定时机制都是大型网站必备的，反向代理一般比较常用，不知道其他应用中的大型网站都还用哪些集群技术？

- [deng1234](#) 发表于: 2007-06-06 13:53:0323 楼 得分:0

-  我来说一下，
1 我们的新闻是从后台添加进去的，新闻添加进去之后并不能在前台显示，

- 等级: 要发布后才能显示，在发布的时候把新闻生成静态页面，如果

在换别的新闻，就必须把以前的新闻撤下来。这样就可以保存发步的新闻是最新的。

2 就技术方面肯定不会用 **hibernate** 的，只是用的 **jsp** 这种最基本的，在写数据的时候一定用的是存储过程。经过测试存储过程的确快很多，

3 数据库用的是 **oracle**。服务器用的是 2 个 **weblogic**

发表于： 2007-06-06 14:21:5624 楼 得分:0

我来说一下，

1 我们的新闻是从后台添加进去的，新闻添加进去之后并不能在前台显示，

要发布后才能显示，在发布的时候把新闻生成静态页面，如果在换别的新闻，就必须把以前的新闻撤下来。这样就可以保存发步的新闻是最新的。

2 就技术方面肯定不会用 **hibernate** 的，只是用的 **jsp** 这种最基本的，在写数据的时候一定用的是存储过程。经过测试存储过程的确快很多，

3 数据库用的是 **oracle**。服务器用的是 2 个 **weblogic**

-

我们也是 **oracle** 存储过程+2 个 **weblogic**，更新机制也几乎一样，看来这个是比较普遍的方法了。

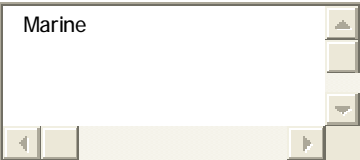
发表于： 2007-06-06 14:28:0225 楼 得分:0

生成静态页面的服务器和 **www** 服务器是两组不同的服务器，页面生成后才会到 **www** 服务器

一部分数据库并不是关系数据库，这样更适合信息衍生

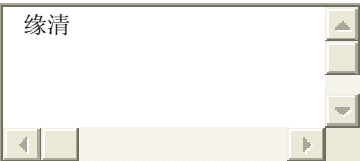
www、**mail** 服务器、路由器多，主要用负载均衡解决访问瓶颈

marine_chen



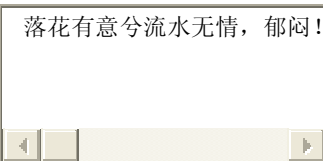
等级:

isline



等级:

didibaba



等级:

发表于： 2007-06-06 14:33:2926 楼 得分:20

网站首页初始化一般都是如何实现？

难道每次打开都拉一次数据？比如各个小模块都怎么关联？

我说的缓存是数据缓存，将当天、热门的数据做成 **hash** 放到内存。页面小模块还是用平常的处理办法来将标题拉出来显示，因为有页面缓存这个东西的存在你不用担心每次会去读数据库。

1、如用户点击，

<http://news.163.com/07/0606/09/3GA0D10N00011229.html>

2、由于使用 url 重写过，实际上可能都是发送到

<http://news.163.com/shownews.jsp?id=3GA0D10N00011229> 去处理。

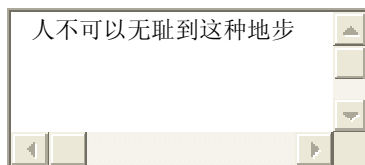
3、shownews.jsp 只需要从内存里面的 hashtable 取得保存的新闻对象即可

4、如内存里面没有，再去读数据库得到新闻

难道每次打开都拉一次数据？

当然不是如此。可以在发布新闻的同时将数据缓存。当然缓存也不会越来越大，在个特定的时间段（如凌晨）剔除过期的数据。

- **net205**

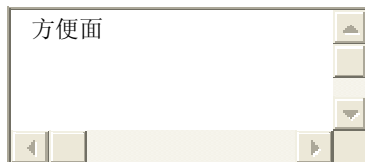


发表于： 2007-06-06 15:03:36 27 楼 得分:0

观望...

- 等级:

- **45Ter**



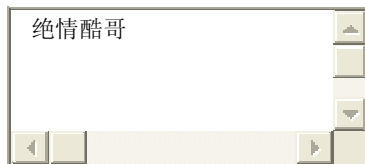
发表于： 2007-06-06 15:09:57 28 楼 得分:0

哇塞，路过，有些技术都没有接触过，向各位学习！

- 等级:

- 发表于： 2007-06-06 15:26:47 29 楼 得分:0

- **cucuchen**



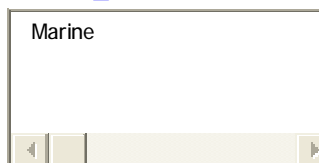
didibaba(落花有意兮流水无情，郁闷!!!)

我严重同意他的观点。

缓存机制可以用 hibernate 实现的那套 ecache,感觉还可以的。

- 等级:

- **marine_chen**



发表于： 2007-06-06 16:04:50 30 楼 得分:0

我说的缓存是数据缓存，将当天、热门的数据做成 hash 放到内存。

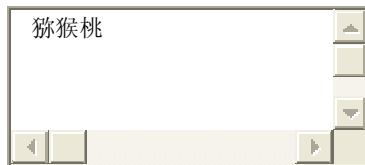
struts、webwork 好像没有像 servlet 的 init 初始化加载数据的功能吧？

- 等级:

页面缓存以前用过 **oscache**，数据缓存用过 **ehcache** 和 **swarmcache**，**ibatis** 还有自带的缓存机制，使用这些缓存加上负载均衡技术实现了一个系统。

经过大家探讨，自己再总结一下，感觉首页这样的设计无非就是静态页定时更新、页面缓存、数据缓存、服务器集群等方法，还有没有更新颖的思路？

- [forevermihoutao](#)



发表于：2007-06-06 16:07:39 31 楼 得分:0

up

-
- 等级：

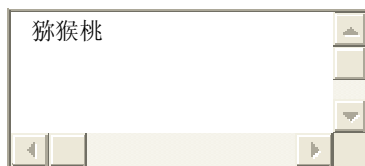
发表于：2007-06-06 16:36:00 32 楼 得分:60

程序开发是一方面，系统架构设计（硬件+网络+软件）是另一方面。

中国的网络分南北电信和网通，访问的 **ip** 就要区分南北进入不同的网络；

然后是集群，包括应用服务器集群和 **web** 服务器集群，应用服务器集群可以采用 **apache+tomcat** 集群和 **weblogic** 集群等，**web** 服务器集群可以用反向代理，也可以用 **NAT** 的方式，或者多域名解析都可以；**Squid** 也可以，反正方法很多，可以根据情况选择；

- [forevermihoutao](#)



软件架构方面，做网站首先需要很多 **web** 服务器存储静态资源，比如图片、视频、静态页等，千万不要把静态资源和应用服务器放在一起；

-
- 等级：

页面数据调用更要认真设计，一些数据查询可以不通过数据库的方式，实时性要求不高的可以使用 **lucene** 来实现，即使有实时性的要求也可以用 **lucene**，**lucene+compass** 还是非常优秀的；

不能用 **lucene** 实现的可以用缓存，分布式缓存可以用 **memcached**，如果有钱的话用 10 来台机器做缓存，>10G 的存储量相信存什么都够了；如果没钱的话可以在页面缓存和数据缓存上下功夫，多用 **OSCACHE** 和 **EHCACHE**，**SWARMCACHE** 也可以，不过据说同步性不是很好；

然后很重要的一点就是数据库，大型网站要用 **oracle**，数据方面操作尽量多用存储过程，绝对提升性能；同时要让 **DBA** 对数据库进行优化，优化后的数据库与没优化的有天壤之别；同时还可以扩展分布式数据库，以后这方面的研究会越来越多；

新闻类的网站可以用静态页存储，采用定时更新机制减轻服务器负担；首页每个小模块可以使用 **oscache** 缓存，这样不用每次都拉数据；

最后是写程序了，一个好的程序员写出来的程序会非常简洁、性能很好，一个初级程序员可能会犯很多低级错误，这也是影响网站性能的原因之一。

I 资料收集：高并发 高性能 高扩展性 Web 2.0 站点架构设计及优化策略

最近专门花时间研究了一下高并发 高性能 高扩展性 Web 2.0 站点架构设计及优化策略，发现了很多不错的资料，继续跟大家分享。——对于期望在大型网络应用的性能测试和性能优化方面获得提高的朋友们来说，尤其应该认真看看。^_^

» 说说大型高并发高负载网站的系统架构 俊麟 [Michael's blog](#)

[bind dlz - 分布式系统的请求分发工具：一个藏袍](#)

bind dlz - 分布式系统的请求分发工具 **bind dlz** 全称是 **bind dynamic loadable zones**，是基于 **bind** 的提供的一个组件，作用看名字就知道了，支持动态域加载支持。 **bind** 已经有很久的历史，目前是搭建 **DNS** 服务器的首选。对于一般网站来说，一个标准的 **bind** 已经完全可以完成所有 **dns** 解决的工作，但在海量域名数量的情况下，**bind** 也确实存在着一些问题： 1、域名解析信息全部存储在文本文件中，这非常容易导致由于编辑出错导致的域名解析出错。 2、**bind** 运行时将全部的解析信息放在内存里，如果数量巨大将可能出现内存不足的情况，同时解析信息重新加载时所耗费的时间也非常值得考虑，由于加载时间较长，所以基本可以不考虑动态的进行域名的调整。 **dlz** 就是为了解决这个问题而针对 **bind** 开发的组件，可以将域名解析信息放在数据库中，从而避免域名信息变动时重新加载的时间，在变动后马上生效。 **dlz** 支持多种数据存储形式，包括文件系统，**Berkeley-DB**，**Postgre-SQL**，**MySQL**，**ODBC**，**LDAP** 等等。性能的比较在这里。 **bind dlz** 这种提供动态的域名调整，并且仍然可

以提供高性能的 dns 解析服务的特点可以应用于提供二级或三级域名服务的分布式系统的前端，对不同的域名解析到所在服务器组上，从而实现可扩展的系统架构。...

[Craigslist 的数据库架构 - DBA notes](#)

[CSDN 视频：CSDN SD 俱乐部与钱宏武探讨如何设计高并发体系架构](#)

[CSDN 视频频道](#)

[Flickr 的开发者的 Web 应用优化技巧 - DBA notes](#)

[YouTube 的架构扩展 - DBA notes](#)

[了解一下 Technorati 的后台数据库架构 - DBA notes](#)

[从 LiveJournal 后台发展看大规模网站性能优化方法: 一个藏袍](#)

从 LiveJournal 后台发展看大规模网站性能优化方法 于敦德 2006-3-16 一、LiveJournal 发展历程 LiveJournal 是 99 年始于校园中的项目，几个人出于爱好做了这样一个应用，以实现以下功能：博客，论坛社会性网络，找到朋友聚合，把朋友的文章聚合在一起 LiveJournal 采用了大量的开源软件，甚至它本身也是一个开源软件。在上线后，LiveJournal 实现了非常快速的增长：2004 年 4 月份：280 万注册用户。2005 年 4 月份：680 万注册用户。2005 年 8 月份：790 万注册用户。达到了每秒钟上千次的页面请求及处理。使用了大量 MySQL 服务器。使用了大量通用组件。二、LiveJournal 架构现状概况 三、从 LiveJournal 发展中学习 LiveJournal 从 1 台服务器发展到 100 台服务器，这其中经历了无数的伤痛，但同时也摸索出了解决这些问题的方法，通过对 LiveJournal 的学习，可以让我们避免 LJ 曾经犯过的错误，并且从一开始就对系统进行良好的设计，以避免后期的痛苦。下面我们一步一步看 LJ 发展的脚步。...

[使用 memcached 进行内存缓存: 一个藏袍](#)

使用 memcached 进行内存缓存 旧文重发 2005.8.9 通常的网页缓存方式有动态缓存和静态缓存等几种，在 ASP.NET 中已经可以实现对页面局部进行缓存，而使用 memcached 的缓存比 ASP.NET 的局部缓存更加灵活，可以缓存任意的对象，不管是否在页面上输出。而 memcached 最大的优点是可以分布式的部署，这对于大规模应用来说也是必不可少的要求。LiveJournal.com 使用了 memcached 在前端进行缓存，取得了良好的效果，而像 wikipedia,sourceforge 等也采用了或即将采用 memcached 作为缓存工具。memcached 可以大规模网站应用发挥巨大的作用。...

[使用开源软件，设计高性能可扩展网站: 一个藏袍](#)

使用开源软件，设计高性能可扩展网站 2006-6-17 于敦德上次我们以 LiveJournal 为例详细分析了一个小网站在一步一步的发展成为大规模的网站中性能优化的方案，以解决在发展中由于负载增长而引起的性能问题，同时在设计网站架构的时候就从根本上避免或者解决这些问题。今天我们来看一下在网站的设计上一些通常使用的解决大规模访问，高负载的方法。我们将主要涉及到以下几方面：1、前端负载 2、业务逻辑层 3、数据层在 LJ 性能优化文章中我们提到对服务器分组是解决

负载问题，实现无限扩展的解决方案。通常中我们会采用类似 LDAP 的方案来解决，这在邮件的服务器以及个人网站，博客的应用中都有使用，在 Windows 下面有类似的 Active Directory 解决方案。有的应用（例如博客或者个人网页）会要求在二级域名解析的时候就将用户定位到所属的服务器群组，这个时候请求还没到应用上面，我们需要在 DNS 里解决这个问题。这个时候可以用到一款软件 bind dlz，这是 bind 的一个插件，用于取代 bind 的文本解析配置文件。它支持包括 LDAP，BDB 在内的多种数据存储方式，可以比较好的解决这个问题。另外一种涉及到 DNS 的问题就是目前普遍存在的南北互联互通的问题，通过 bind9 内置的视图功能可以根据不同的 IP 来源解析出不同的结果，从而将南方的用户解析到南方的服务器，北方的用户解析到北方的服务器。这个过程中会碰到两个问题，一是取得南北 IP 的分布列表，二是保证南北服务器之间的通讯顺畅。第一个问题有个笨办法解决，从日志里取出所有的访问者 IP，写一个脚本，从南北的服务器分别 ping 回去，然后分析结果，可以得到一个大致准确的列表，当然最好的办法还是直到从运营商那里拿到这份列表 (update: 参见这篇文章)。后一个问题解决办法比较多，最好的办法就是租用双线机房，同一台机器，双 IP，南北同时接入，差一些的办法就是南北各自找机房，通过大量的测试找出中间通讯顺畅的两个机房，后一种通常来说成本较低，但效果较差，维护不便。另外 DNS 负载均衡也是广泛使用的一种负载均衡方法，通过并列的多条 A 记录将访问随即的分布到多台前端服务器上，这种通常使用在静态页面居多的应用上，几大门户内容部分的前端很多都是用的这种方法。用户被定位到正确的服务器群组后，应用程序就接手用户的请求，并开始沿着定义好的业务逻辑进行处理。这些请求主要包括两类静态文件(图片, js 脚本, css 等)，动态请求。静态请求一般使用 squid 进行缓存处理，可以根据应用的规模采用不同的缓存配置方案，可以是一级缓存，也可以是多级缓存，一般情况下 cache 的命中率可以达到 70% 左右，能够比较有效的提升服务器处理能力。Apache 的 deflate 模块可以压缩传输数据，提高速度，2.0 版本以后的 cache 模块也内置实现磁盘和内存的缓存，而不必要一定做反向代理。动态请求目前一般有两种处理方式，一种是静态化，在页面发生变化时重新静态页面，现在大量的 CMS，BBS 都采用这种方案，加上 cache，可以提供较快的访问速度。这种通常是写操作较少的应用比较适合的解决方案。另一种解决办法是动态缓存，所有的访问都仍然通过应用处理，只是应用处理的时候会更多的使用内存，而不是数据库。通常访问数据库的操作是极慢的，而访问内存的操作很快，至少是一个数量级的差距，使用 memcached 可以实现这一解决方案，做的好的 memcache 甚至可以达到 90% 以上的缓存命中率。10 年前我用的还是 2M 的内存，那时的一本杂事上曾经风趣的描述一对父子的对话： 儿子：爸爸，我想要 1G 的内存。爸爸：儿子，不行，即使是你过生日也不行。时至今日，大内存的成本已经完全可以承受。Google 使用了大量的 PC 机建立集群用于数据处理，而我一直觉得，使用大内存 PC 可以很低成本的解决前端甚至中间的负载问题。由于 PC 硬盘寿命比较短，速度比

较慢，CPU 也稍慢，用于做 web 前端既便宜，又能充分发挥大内存的优势，而且坏了的话只需要替换即可，不存在数据的迁移问题。下面就是应用的设计。应用在设计的时候应当尽量设计成支持可扩展的数据库设计，数据库可以动态的添加，同时支持内存缓存，这样的成本是最低的。另外一种应用设计的方法是采用中间件，例如 ICE。这种方案的优点是前端应用可以设计的相对简单，数据层对于前端应用透明，由 ICE 提供，数据库分布式的设计在后端实现，使用 ICE 封装后给前端应用使用，这路设计对每一部分设计的要求较低，将业务更好的分层，但由于引入了中间件，分了更多层，实现起来成本也相对较高。在数据库的设计上一方面可以使用集群，一方面进行分组。同时在细节上将数据库优化的原则尽量应用，数据库结构和数据层应用在设计上尽量避免临时表的创建、死锁的产生。数据库优化的原则在网上比较常见，多 google 一下就能解决问题。在数据库的选择上可以根据自己的习惯选择，Oracle，MySQL 等，并非 Oracle 就够解决所有的问题，也并非 MySQL 就代表小应用，合适的就是最好的。前面讲的都是基于软件的性能设计方案，实际上硬件的良好搭配使用也可以有效的降低时间成本，以及开发维护成本，只是在这里我们不再展开。网站架构的设计是一个整体的工程，在设计的时候需要考虑到性能，可拓展性，硬件成本，时间成本等等，如何根据业务的定位，资金，时间，人员的条件设计合适的方案是件比较困难的事情，但多想多实践，终究会建立一套适合自己的网站设计理念，用于指导网站的设计工作，为网站的发展奠定良好的基础。...

初创网站与开源软件: 一个藏袍

初创网站与开源软件前面有一篇文章中提到过开源软件，不过主要是在系统运维的角度去讲的，主要分析一些系统级的开源软件(例如 bind, memcached)，这里我们讨论的是用于搭建初创网站应用的开源软件(例如 phpbb, phparticle)，运行在 Linux，MySQL，Apache, PHP, Java 等下面。创业期的网站往往采用比较简单的系统架构，或者是直接使用比较成熟的开源软件。使用开源软件的好处是搭建速度快，基本不需要开发，买个空间域名，下个软件一搭建，用个半天就搞定了，一个崭新的网站就开张了，在前期可以极大程度的节约时间成本和开发成本。当然使用开源软件搭建应用也存在一些局限性，这是我们要重点研究的，而研究的目的就是如何在开源软件选型时以及接下来的维护过程中尽量避免。一方面是开源软件一般只有在比较成熟的领域才有，如果是一些创新型的项目很难找到合适的开源软件，这个时候没什么好的解决办法，如果非要用开源的话一般会找一个最相似的改一下。实际上目前开源的项目也比较多了，在 sf.net 上可以找到各种各样的开源项目。选型的时候尽量应该选取一个程序架构比较简单的，不一定越简单越好，但一定要简单，一目了然，别用什么太高级的特性，互联网应用项目不需要太复杂的框架。原因有两个，一个是框架复杂无非是为了实现更好的可扩展性和更清晰的层次，而我们正在做的互联网应用范围一般会比开源软件设计时所考虑的范围小的多，所以有的应用会显得设计过度，另外追求完美的层次划分导致的太复杂的继承派生关系也会影响到整

个系统维护的工作量。建议应用只需要包含三个层就可以了，数据(实体)层，业务逻辑层，表现层。太复杂的设计容易降低开发效率，提高维护成本，在出现性能问题或者突发事件的时候也不容易找到原因。另外一个问题是开源软件的后期维护和继续开发可能会存在问题，这一点不是绝对的，取决于开源软件的架构是否清晰合理，扩展性好，如果是较小的改动可能一般不存在什么问题，例如添加一项用户属性或者文章属性，但有些需求可能就不是很容易实现了。例如网站发展到一定阶段后可能会考虑扩展产品线，原来只提供一个论坛加上 cms，现在要再加上商城，那用户系统就会有问题，如何解决这个问题已经不仅仅是改一下论坛或者 cms 就可以解决了，这个时候我们需要上升到更高的层次来考虑问题，是否需要建立针对整个网站的用户认证系统，实现单点登录，用户可以在产品间无缝切换而且保持登录状态。由于网站初始的用户数据可能大部分都存放在论坛里，这个时候我们需要把用户数据独立出来就会碰到麻烦，如何既能把用户数据独立出来又不影响论坛原有系统的继续运行会是件很头痛的事情。经过一段时间的运行，除非是特别好的设计以及比较好的维护，一般都会在论坛里存在各种各样乱七八糟的对用户信息的调用，而且是直接针对数据库的，这样如果要移走的话要修改代码的工作量将不容忽视，而另外一个解决办法是复制一份用户数据出来，以新的用户数据库为主，论坛里的用户数据通过同步或异步的机制实现同步。最好的解决办法就是在选型时选一个数据层封装的比较好的，sql 代码不要到处飞的软件，然后在维护的时候保持系统原有的优良风格，把所有涉及到数据库的操作都放到数据层或者实体层里，这样无论对数据进行什么扩展，代码修改起来都比较方便，基本不会对上层的代码产生影响。网站访问速度问题对初创网站来说一般考虑的比较少，买个空间或者托管服务器，搭建好应用后基本上就开始运转了，只有到真正面临极大的速度访问瓶颈后才会真正对这个问题产生重视。实际上在从网站的开始阶段开始，速度问题就会一直存在，并且会随着网站的发展也不断演进。一个网站最基本的要求，就是有比较快的访问速度，没有速度，再好的内容或服务也出不来。所以，访问速度在网站初创的时候就需要考虑，无论是采用开源软件还是自己开发都需要注意，数据层尽量能够正确，高效的使用 SQL。SQL 包含的语法比较复杂，实现同样一个效果如果考虑到应用层的的不同实现方法，可能有好几种方法，但里面只有一种是最高效的，而通常情况下，高效的 SQL 一般是那个最简单的 SQL。在初期这个问题可能不是特别明显，当访问量大起来以后，这个可能成为最主要的性能瓶颈，各种杂乱无章的 SQL 会让人看的疯掉。当然前期没注意的话后期也有解决办法，只不过可能不会解决的特别彻底，但还是要吧非常有效的提升性能。看 MySQL 的 SlowQuery Log 是一个最为简便的方法，把执行时间超过 1 秒的查询记录下来，然后分析，把该加的索引加上，该简单的 SQL 简化。另外也可以通过 Showprocesslist 查看当前数据库服务器的死锁进程，从而锁定导致问题的 SQL 语句。另外在数据库配置文件上可以做一些优化，也可以很好的提升性能，这些文章在网站也比较多，这里就不展开。

这些工作都做了以后，下面数据库如果再出现性能问题就需要考虑多台服务器了，一台服务器已经解决不了问题了，我以前的文章中也提到过，这里也不再展开。其它解决速度问题的办法就不仅仅是在应用里面就可以实现的了，需要从更高的高度去设计系统，考虑到服务器，网络的架构，以及各种系统级应用软件的配合，这里也不再展开。良好设计并实现的应用+中间件+良好的分布式设计的数据库+良好的系统配置+良好的服务器/网络结构，就可以支撑起一个较大规模的网站了，加上前面的几篇文章，一个小网站发展到大网站的过程基本上就齐了。这个过程会是一个充满艰辛和乐趣的过程，也是一个可以逐渐过渡的过程，主动出击，提前考虑，减少救火可以让这个过程轻松一些。...

大型 web2.0 互动网站设计方案 - jim_yeejee 的专栏 - CSDNBlog

大型 SNS 互动网站实现方案，大型 web2.0 互动网站实现方案

大型 Web2.0 站点构建技术初探 - guxianga - CSDNBlog

缓存区还为那些不需要记入数据库的数据提供了驿站，比如为跟踪用户会话而创建的临时文件--Benedetto 坦言他需要在这方面补课，

高并发高流量网站架构

首先在整个网络的高度讨论了使用 cdn，镜像，以及 DNS 区域解析等技术对负载均衡带来的便利及各自的优缺点比较。然后在局域网层次对第四层交换技术，包括硬件解决方案 F5 和软件解决方案 LVS，进行了探讨和比较。再次在在单服务器端，本文着重讨论了单台服务器的 socket 优化，硬盘级缓存技术，内存级缓存技术，cpu 与 io 平衡技术（即以运算为主的程序与以数据读写为主的程序搭配部署），读写分离技术等。在应用层，本文介绍了一些企业常用的技术，以及选择使用该技术的理由。本文选取有代表性的网站服务器程序，数据库程序，数据表存储

高性能网站性能优化 - 人月 - CSDNBlog

网站负载均衡

转 memcache 在 blog 系统中的应用 ★★★★★ ?

<http://hi.baidu.com/gowtd/blog/item/4fa90f2353f5444e935807fa.html>

对 memcache 的接触也就不到 4-5 天的时间，大半的时间是花在研究如何利用 memcache 的接口，用简单有效的方式融入到我们 blog 应用系统中。基于此前日志模块已经使用 memcache，并且在实际测试中有良好的性能表现（很高的 cache 命中率，跟总体数量少有很大关系），相册模块也开始在 dao 中提供对 memcache 的支持。只是原先封装 memcache 接口实在是够抽象，日志模块的 dao 使用 memcache 就使得代码量增加了 50%，而且代码也显得非常凌乱。在应用到逻辑更加复杂的相册模块时，dao 层变得更加庞大；并且由于对 memcache 还不够了解，使得相册的 dao 错误百出。可以说，相册支持 memcache 的初试版本在关闭 memcache 后，性能会有一半的下降；打开 memcache 后性能不会有什么增长，反而会有些下降。而且代码更加凌乱，可读性和可维护性很差。没有办法，只有亲自来重新写一下相册的 dao。通过周四和周五期间和 taotao 同志的多次协商和讨论，总算把新的 dao 给完成了。代码精简了不少，memcache 的接口也有了一定的抽象（虽然还是比较丑陋的接口），基本

每个 **dao** 接口都可以在 10—15 行代码里面搞定。可读性大大提高，对二级索引等比较烦琐的管理也都被屏蔽了起来。下礼拜需要把他们放到服务器上去测试一下，当然，还要逼着 **taotao** 把接口搞的更好看些。

今天休息，看了下 **memcache** 的 **client** 和 **server** 的代码，很简单的东西。但是在这中间，我还是觉得有些东西值得思考。

1. **memcache** 对 **java** 对象的支持。**memcache** 的客户端对 **java** 对象的支持做了些优化。主要是对 **primitive** 类型的优化。**memcache** 服务器对所存储的数据类型是完全无知的，甚至也不是一个对象存储系统。它所能看到的就是一块块装着数据的内存。而在 **memcache** 的客户端，如果完全按照 **java** 的 **oo** 思想来把对象放进去，还是有些低效的。比如把一个 **boolean** 值要放进 **cache** 里去，**java** 客户端的通常做法是生成一个 **Boolean** 对象，然后把这个对象串行化，在写入到服务器上。这就浪费了服务器的存储内存。一个 **Boolean** 对象好歹也需要 40byte 的内存，服务器要分配给它 64byte 的内存来用于存储该对象。而一个 **boolean** 值实际上只要 1 个 **byte** 就能搞定的事情。**memcache** 这点做的比较好，它对 **primitive** 类型做压缩，同样是 **boolean** 值，用 2 个 **byte** 来存储值并写给服务器。前面一个 **byte** 存储值类型，后面一个 **byte** 存储实际的值。这个还是很值得推崇的方法。

对其他对象的存储支持，**memcache** 就采取通用的对象序列化方法，到取回对象时，再重建这个对象。这种方法的好处就是简单，程序员不需要考虑对象的重建问题，依赖 **java** 的特性来重建对象。但是我认为，在我们的应用系统中，使用这种方法是不必要的，可以用其他有效的方式来提高效率和性能。举个例子，一个 **Photo** 对象要放到 **cache** 里面取，假设经过序列化后对象大小变成 139byte（很普遍的，实际更大）。根据 **memcache** 的内存分层分配，就要实际分配 256byte 的实际内存给它。如果对象大小是 260byte，系统就要分配 512byte 的内存给它。其中将近一半的内存是用不上的。而真的而去看序列化后的对象，里面很多信息都是用来在 **java** 重建对象时用的。如果我们也能参照 **memcache** 对 **primitive** 支持，让 **Photo** 对象实现一个特定的接口，这个接口能从一个字串中初试化一个 **photo** 对象来。这样就省了在服务器上存储 **Photo** 对象的很多类型信息，节省了对内存占用。但是，这就需要我们所有放入到 **cache** 中的对象都要实现这样的接口，限制了 **memcache** 的通用性。不过，在我们的 **blog** 应用中，没有多少种对象会放到 **cache** 中，这种通用性可以牺牲一下。

2. 在 **blog** 系统中，将 **blog** 对象和 **Photo** 对象同时存储，是否合适。在我看来，**Blog** 对象对于 **Photo** 对象来说完全是个大家伙。一个包含长篇大论的 **blog** 对象，要占用很大一块内存，而且每个 **blog** 对象的大小还不一定，或大或小。而 **Photo** 对象则大小相对稳定。根据 **memcache** 的 **slab** 内存分配原则，当内存已经无法再分配时，要根据所请求放入的对象的大小到所对应的 **slab** 上以 **LRU** 算法把一些对象交换出去。可以设想一下，如果在一个 **cache** 服务器上，有很多小的 **Photo** 对象，和一些大的 **Blog** 对象。并且在开始时，**cache** 服务器先很频繁地为 **Photo** 对象提供存储服务。很明显，当系统稳定时，新放入 **Blog** 对象更加容易引起某个 **slab** 上的对象被交换。因为系统中的大块内存都被无数小的 **Photo** 对象所分割占用，而 **Blog** 对象只能获得一小部分的内存。此时，系统不会调整 **Photo** 对象占用的内存来补充 **Blog** 对象，因为两者很大程度上是处在两个不同的 **slab** 上。可以说，在 **cache** 服务器上，**blog** 对象的平均生命周期会比 **Photo** 短，更容易被交换出去。从而造成 **blog** 对象的失配比率会比 **photo** 对象要高。我的想法是将 **blog** 对象和 **photo** 对象分别存放，让一群大小基本相同的对象放置在一个 **cache** 服务器上，可能是比较好点。这可以通过在选择 **cache** 存储服务器时，同时考虑对象的大小来完成

I CommunityServer 性能问题浅析

前言

可能有很多朋友在使用 **CommunityServer**(以下简称 **CS**)的过程中, 当数据越来越多后, 速度会越来越慢, 资源耗用越来越大, 对于性能不好的服务器, 简直像一场噩梦一样, 我终于刚刚结束了这个噩梦, 简单谈谈是什么原因导致了 **CS** 在性能上存在的种种问题。(我对于数据库方面不是很专业, 所以如果本文中有何谬误, 敬请各位指出, 不胜感谢!)

忘了自我介绍一下, 我是宝玉, 以前做过 **Asp.net Forums** 和 **CommunityServer** 的本地化工作, 母校西工大的民间社区 (<http://www.openlab.net.cn>)用的是 **CS** 系统。该有人骂我做广告了, 其实我是防盗版, 郭安定大哥那学的, 哈哈!

性能问题分析

鸡肋式的多站点支持

其中一个性能影响就是它的多站点功能, 也许这确实是个不错的注意: 同一个数据库, 不同域名就可以有完全独立的站点, 但是对于绝大部分用户来说, 这个真的有用么? 首先姑且不讨论它是否真的那么有用, 但是在性能上, 他绝对会有一定影响的: 系统初始化的时候, 首先要加载所有的站点设置, 这也是为什么 **CS** 第一次访问会那么慢的原因之一; 然后大部分查询的时候, 都要带上 **SettingId** 字段, 并且在数据库中, 对这个字段的索引并没有建的很理想, 对于大量数据的查询来说, 如果没有合理的建索引, 有时候多一个查询条件对于性能会带来极大的影响。

内容数据的集中式存储

一般的系统, 都尽可能的将大量的内容数据分开存储(例如飞信系统的用户存储, 就是分库的^_^), 对于数据库, 更是有专门的分库方案, 这都是为了增加性能, 提高检索效率。而 **CS** 由于架构的原因, 将论坛、博客、相册、留言板等内容管理相关的信息, 全部保存在 **cs_Groups**(分组)、**cs_Section**(分类)、**cs_Threads**(主题索引)、**cs_Posts**(内容数据), 这种架构给代码编写上带来了极大的便利, 但是在性能上, 不折不扣是个性能杀手, 这也是 **CS** 慢的最根本原因, 举个例子, 假如我的论坛有 **100W** 数据, 博客有 **5** 万条数据、相册有 **10** 万条数据, 如果我要检索最新博客帖子, 那么我要去这 **120** 万数据里面检索符合条件的数据, 并且要加上诸如 **SettingsId**、**ApplicationType** 等用来区分属于哪个站点, 哪种数据类型之类的条件, 数据一多, 必然会是一场噩梦, 让你的查询响应速度越来越慢, 从几秒钟到几十秒钟到 **Sql** 超时。

过于依赖缓存

缓存是个很好的东西, 可以大大的减少数据库的访问, 是 **asp.net** 程序提高性能必不可少的。不知道各位在设计开发系统, 用缓存用的很爽的时候, 有没有想过, 如果缓存失效了会怎么样? 如果缓存太大了会怎么样? 相信各位 **CS** 会有一个感觉, 那就是 **CS** 刚启动的时候速度好慢, 或者使用过程中突然变的很慢, 那就是因为好多数据还没有初始化到缓存, 例如站点设置、用户资料、**Groups** 集合、**Setions** 集合等等一系列信息, 这一系列信息的加载加起来在服务器性能不够好的情况下是个漫长的过程, 如果碰巧还要去查询最新论坛帖子、未回复的帖子之类, 那么噩梦就开始了, 这时候就要拼人品了, 看你是不是应用程序池刚重启完的第一个人 **o(n_n)o**。**CS** 在缓存的策略上, 细粒度不够, 一般都是一个集合一个集合的进行缓存(例如最新论坛帖子集合), 这样导致缓存需要频繁更新, 而且缓存内的数据一般比较大, 内存占用涨的很快, 内存涨的快又导致了应用程序池频繁重启, 这样, **CS** 在缓存方面的优势反而变成了一种缺陷, 导致服务器的资源占用居高不下。

CCS 的雪上加霜

前面说过，我做过 CS 的本地化开发，加了不少 CS 的本地化开发工作，但是由于当时数据库知识的匮乏，导致了一些在性能上雪上加霜的行为，例如精华帖子功能，其中标志是否为精华帖(精华等级)的 **ValuedLevel** 字段没有加上索引，在数据量大的情况下，检索会比较慢。由于我已经不在做 CCS 的开发，已经没有办法来修正这些性能问题了，只能对大家表示歉意。

后记

如何解决？

最简单就是等着升级了，相信 CS 以后的版本会越来越强劲的，这些问题肯定会逐步解决的。如果等不及的话，就只能自己动手了，使用 **Sql Profiler** 监测 **Sql** 的执行，找出影响性能的查询，然后针对性优化。

前面我说我结束 CS 性能的噩梦，肯定有朋友会问我怎么结束的了，在此，就先埋一个伏笔了，在 05 年的时候，我就开始如何构思开发一套高性能的类似 CS 的系统，06 年初开始设计，然后利用业余时间进行了具体的开发，到今天已经有了小成，在性能上有了质的飞跃，针对这套系统的设计和性能优化的心得，我会逐渐以博客的形式来和大家一起分享交流。

I Digg PHP's Scalability and Performance

• [listen](#) 

Monday April 10, 2006 9:28AM

by [Brian Fioca](#) in [Technical](#)

Several weeks ago there was a [notable bit of controversy](#) over some comments made by James Gosling, father of the Java programming language. He has since [addressed the flame war](#) that erupted, but the whole ordeal got me thinking seriously about PHP and its scalability and performance abilities compared to Java. I knew that several hugely popular Web 2.0 applications were written in scripting languages like PHP, so I contacted Owen Byrne - Senior Software Engineer at [digg.com](#) to learn how he addressed any problems they encountered during their meteoric growth. This article addresses the all-to-common false assumptions about the cost of scalability and performance in PHP applications.

At the time Gosling's comments were made, I was working on tuning and optimizing the source code and server configuration for the launch of [Jobby](#), a Web 2.0 resume tracking application written using the [WASP PHP framework](#). I really hadn't done any substantial research on how to best optimize PHP applications at the time. My background is heavy in the architecture and development of highly scalable applications in Java, but I realized there were enough substantial differences between Java and PHP to cause me concern. In my experience, it was certainly faster to develop web applications in languages like PHP; but I was curious as to how much of that time savings might be lost to performance tuning and scaling costs. What I found was both encouraging and surprising.

What are Performance and Scalability?

Before I go on, I want to make sure the ideas of performance and scalability are understood. Performance is measured by the output behavior of the application. In other words, performance is whether or not the app is fast. A good performing web application is expected to render a page in around or under 1 second (depending on the complexity of the page, of

course). Scalability is the ability of the application to maintain good performance under heavy load with the addition of resources. For example, as the popularity of a web application grows, it can be called scalable if you can maintain good performance metrics by simply making small hardware additions. With that in mind, I wondered how PHP would perform under heavy load, and whether it would scale well compared with Java.

Hardware Cost

My first concern was raw horsepower. Executing scripting language code is more hardware intensive because the code isn't compiled. The hardware we had available for the launch of Jobby was a single hosted Linux server with a 2GHz processor and 1GB of RAM. On this single modest server I was going to have to run both Apache 2 and MySQL. Previous applications I had worked on in Java had been deployed on 10-20 application servers with at least 2 dedicated, massively parallel, ultra expensive database servers. Of course, these applications handled traffic in the millions of hits per month.

To get a better idea of what was in store for a heavily loaded PHP application, I set up an interview with Owen Byrne, cofounder and Senior Software Engineer at digg.com. From talking with Owen I learned digg.com gets on the order of 200 million page views per month, and they're able to handle it with only 3 web servers and 8 small database servers (I'll discuss the reason for so many database servers in the next section). Even better news was that they were able to handle their first year's worth of growth on a single hosted server like the one I was using. My hardware worries were relieved. The hardware requirements to run high-traffic PHP applications didn't seem to be more costly than for Java.

Database Cost

Next I was worried about database costs. The enterprise Java applications I had worked on were powered by expensive database software like Oracle, Informix, and DB2. I had decided early on to use MySQL for my database, which is of course free. I wondered whether the simplicity of MySQL would be a liability when it came to trying to squeeze the last bit of performance out of the database. MySQL has had a reputation for being slow in the past, but most of that seems to have come from sub-optimal configuration and the overuse of MyISAM tables. Owen confirmed that the use of InnoDB for tables for read/write data makes a massive performance difference.

There are some scalability issues with MySQL, one being the need for large amounts of slave databases. However, these issues are decidedly not PHP related, and are being addressed in future versions of MySQL. It could be argued that even with the large amount of slave databases that are needed, the hardware required to support them is less expensive than the 8+ CPU boxes that typically power large Oracle or DB2 databases. The database requirements to run massive PHP applications still weren't more costly than for Java.

PHP Coding Cost

Lastly, and most importantly, I was worried about scalability and performance costs directly attributed to the PHP language itself. During my conversation with Owen I asked him if there were any performance or scalability problems he encountered that were related to having

chosen to write the application in PHP. A bit to my surprise, he responded by saying, “none of the scaling challenges we faced had anything to do with PHP,” and that “the biggest issues faced were database related.” He even added, “in fact, we found that the lightweight nature of PHP allowed us to easily move processing tasks from the database to PHP in order to deal with that problem.” Owen mentioned they use the [APC](#) PHP accelerator platform as well as [MCache](#) to lighten their database load. Still, I was skeptical. I had written Jobby entirely in PHP 5 using a framework which uses a highly object oriented MVC architecture to provide application development scalability. How would this hold up to large amounts of traffic?

My worries were largely related to the PHP engine having to effectively parse and interpret every included class on each page load. I discovered this was just my misunderstanding of the best way to configure a PHP server. After doing some research, I found that by using a combination of Apache 2’s worker threads, FastCGI, and a PHP accelerator, this was no longer a problem. Any class or script loading overhead was only encountered on the first page load. Subsequent page loads were of comparative performance to a typical Java application. Making these configuration changes were trivial and generated massive performance gains. With regard to scalability and performance, PHP itself, even PHP 5 with heavy OO, was not more costly than Java.

Conclusion

Jobby was launched successfully on its single modest server and, thanks to links from [Ajaxian](#) and [TechCrunch](#), went on to happily survive hundreds of thousands of hits in a single week. Assuming I applied all of my new found PHP tuning knowledge correctly, the application should be able to handle much more load on its current hardware.

Digg is in the process of preparing to scale to 10 times current load. I asked Owen Byrne if that meant an increase in headcount and he said that wasn’t necessary. The only real change they identified was a switch to a different database platform. There doesn’t seem to be any additional manpower cost to PHP scalability either.

It turns out that it really is fast *and* cheap to develop applications in PHP. Most scaling and performance challenges are almost always related to the data layer, and are common across all language platforms. Even as a self-proclaimed PHP evangelist, I was very startled to find out that all of the theories I was subscribing to were true. There is simply no truth to the idea that Java is better than scripting languages at writing scalable web applications. I won’t go as far as to say that PHP is better than Java, because it is never that simple. However it just isn’t true to say that PHP doesn’t scale, and with the rise of Web 2.0, sites like [Digg](#), [Flickr](#), and even [Jobby](#) are proving that large scale applications can be rapidly built and maintained on-the-cheap, by one or two developers.

Further Reading

I YouTube Architecture



Tue, 07/17/2007 - 20:20 — [Todd Hoff](#)

[YouTube Architecture \(3936\)](#)

YouTube grew incredibly fast, to over 100 million video views per day, with only a handful of people responsible for scaling the site. How did they manage to deliver all that video to all those users? And how have they evolved since being acquired by Google?

Information Sources

- [Google Video](#)

Platform

- [Apache](#)
- [Python](#)
- [Linux](#) (SuSe)
- [MySQL](#)
- [psyco](#), a dynamic python->C compiler
- [lighttpd](#) for video instead of Apache

What's Inside?

The Stats

- Supports the delivery of over 100 million videos per day.
- Founded 2/2005
- 3/2006 30 million video views/day
- 7/2006 100 million video views/day

- 2 sysadmins, 2 scalability software architects
- 2 feature developers, 2 network engineers, 1 DBA

Recipe for handling rapid growth

```
while (true)
{
    identify_and_fix_bottlenecks();
    drink();
    sleep();
    notice_new_bottleneck();
}
```

This loop runs many times a day.

Web Servers

- NetScaler is used for load balancing and caching static content.
- Run Apache with mod_fast_cgi.
- Requests are routed for handling by a Python application server.
- Application server talks to various databases and other information sources to get all the data and formats the html page.
- Can usually scale web tier by adding more machines.
- The Python web code is usually NOT the bottleneck, it spends most of its time blocked on RPCs.
- Python allows rapid flexible development and **deployment**. This is critical given the competition they face.
- Usually less than 100 ms page service times.

- Use psyco, a dynamic python->C compiler that uses a JIT compiler approach to optimize inner loops.
- For high CPU intensive activities like encryption, they use C extensions.
- Some pre-generated cached HTML for expensive to render blocks.
- Row level caching in the database.
- Fully formed Python objects are cached.
- Some data are calculated and sent to each application so the values are cached in local memory. This is an underused strategy. The fastest cache is in your application server and it doesn't take much time to send precalculated data to all your servers. Just have an agent that watches for changes, precalculates, and sends.

Video Serving

- Costs include bandwidth, hardware, and power consumption.
- Each video hosted by a mini-cluster. Each video is served by more than one machine.
- Using a cluster means:
 - More disks serving content which means more speed.
 - Headroom. If a machine goes down others can take over.
 - There are online backups.
- Servers use the lighttpd web server for video:
 - Apache had too much overhead.
 - Uses **epoll** to wait on multiple fds.
 - Switched from single process to multiple process configuration to handle more connections.
- Most popular content is moved to a **CDN** (content delivery network):
 - CDNs replicate content in multiple places. There's a better chance of content being closer to the user, with fewer hops, and content will run over a more friendly network.
 - CDN machines mostly serve out of memory because the content is so popular there's

little thrashing of content into and out of memory.

- Less popular content (1-20 views per day) uses YouTube servers in various colo sites.
- There's a long tail effect. A video may have a few plays, but lots of videos are being played. Random disks blocks are being accessed.
- **Caching** doesn't do a lot of good in this scenario, so spending money on more cache may not make sense. This is a very interesting point. If you have a long tail product caching won't always be your performance savior.
- Tune **RAID** controller and pay attention to other lower level issues to help.
- Tune memory on each machine so there's not too much and not too little.

Serving Video Key Points

- Keep it simple and cheap.
- Keep a simple network path. Not too many devices between content and users. Routers, switches, and other appliances may not be able to keep up with so much load.
- Use commodity hardware. More expensive hardware gets the more expensive everything else gets too (support contracts). You are also less likely find help on the net.
- Use simple common tools. They use most tools build into Linux and layer on top of those.
- Handle random seeks well (SATA, tweaks).

Serving Thumbnails

- Surprisingly difficult to do efficiently.
- There are a like 4 thumbnails for each video so there are a lot more thumbnails than videos.
- Thumbnails are hosted on just a few machines.
- Saw problems associated with serving a lot of small objects:
 - Lots of disk seeks and problems with inode caches and page caches at OS level.
 - Ran into per directory file limit. Ext3 in particular. Moved to a more hierarchical

structure. Recent improvements in the 2.6 kernel may improve Ext3 large directory handling up to **100 times**, yet storing lots of files in a file system is still not a good idea.

- A high number of requests/sec as web pages can display 60 thumbnails on page.
- Under such high loads Apache performed badly.
- Used squid (reverse proxy) in front of Apache. This worked for a while, but as load increased performance eventually decreased. Went from 300 requests/second to 20.
- Tried using lighttpd but with a single threaded it stalled. Run into problems with multiprocesses mode because they would each keep a separate cache.
- With so many images setting up a new machine took over 24 hours.
- Rebooting machine took 6-10 hours for cache to warm up to not go to disk.
- To solve all their problems they started using Google's **BigTable**, a distributed data store:
 - Avoids small file problem because it clumps files together.
 - Fast, fault tolerant. Assumes its working on a unreliable network.
 - Lower latency because it uses a distributed multilevel cache. This cache works across different collocation sites.
 - For more information on BigTable take a look at [Google Architecture](#), [GoogleTalk Architecture](#), and [BigTable](#).

Databases

- The Early Years
 - Use MySQL to store meta data like users, tags, and descriptions.
 - Served data off a monolithic RAID 10 Volume with 10 disks.
 - Living off credit cards so they leased hardware. When they needed more hardware to handle load it took a few days to order and get delivered.
 - They went through a common evolution: single server, went to a single master with multiple read slaves, then partitioned the database, and then settled on a sharding

approach.

- Suffered from replica lag. The master is multi-threaded and runs on a large machine so it can handle a lot of work. Slaves are single threaded and usually run on lesser machines and replication is asynchronous, so the slaves can lag significantly behind the master.

- Updates cause cache misses which goes to disk where slow I/O causes slow replication.

- Using a replicating architecture you need to spend a lot of money for incremental bits of write performance.

- One of their solutions was prioritize traffic by splitting the data into two clusters: a video watch pool and a general cluster. The idea is that people want to watch video so that function should get the most resources. The social networking features of YouTube are less important so they can be routed to a less capable cluster.

- The later years:

- Went to database partitioning.

- Split into shards with users assigned to different shards.

- Spreads writes and reads.

- Much better cache locality which means less IO.

- Resulted in a 30% hardware reduction.

- Reduced replica lag to 0.

- Can now scale database almost arbitrarily.

Data Center Strategy

- Used managed hosting providers at first. Living off credit cards so it was the only way.

- Managed hosting can't scale with you. You can't control hardware or make favorable networking agreements.

- So they went to a colocation arrangement. Now they can customize everything and negotiate their own contracts.

- Use 5 or 6 data centers plus the CDN.

- Videos come out of any data center. Not closest match or anything. If a video is popular enough it will move into the CDN.
- Video bandwidth dependent, not really latency dependent. Can come from any colo.
- For images latency matters, especially when you have 60 images on a page.
- Images are replicated to different data centers using BigTable. Code looks at different metrics to know who is closest.

Lessons Learned

- **Stall for time.** Creative and risky tricks can help you cope in the short term while you work out longer term solutions.
- **Prioritize.** Know what's essential to your service and prioritize your resources and efforts around those priorities.
- **Pick your battles.** Don't be afraid to outsource some essential services. YouTube uses a CDN to distribute their most popular content. Creating their own network would have taken too long and cost too much. You may have similar opportunities in your system. Take a look at [Software as a Service](#) for more ideas.
- **Keep it simple!** Simplicity allows you to rearchitect more quickly so you can respond to problems. It's true that nobody really knows what simplicity is, but if you aren't afraid to make changes then that's a good sign simplicity is happening.
- **Shard.** Sharding helps to isolate and constrain storage, CPU, memory, and IO. It's not just about getting more writes performance.
- **Constant iteration on bottlenecks:**
 - Software: DB, caching
 - OS: disk I/O
 - Hardware: memory, RAID
- **You succeed as a team.** Have a good cross discipline team that understands the

whole system and what's underneath the system. People who can set up printers, machines, install networks, and so on. With a good team all things are possible.

1. [Jesse](#) • [Comments \(78\)](#) • [April 10th](#)

Justin Silverton at [Jaslabs](#) has a supposed list of [10 tips for optimizing MySQL queries](#). I couldn't read this and let it stand because this list is really, really bad. Some [guy named Mike](#) noted this, too. So in this entry I'll do two things: first, I'll explain why his list is bad; second, I'll present my own list which, hopefully, is much better. Onward, intrepid readers!

Why That List Sucks

1. He's swinging for the top of the trees

The rule in any situation where you want to optimize some code is that you first profile it and then find the bottlenecks. Mr. Silverton, however, aims right for the tippy top of the trees. I'd say 60% of database optimization is properly understanding SQL and the basics of databases. You need to understand joins vs. subselects, column indices, how to normalize data, etc. The next 35% is understanding the performance characteristics of your database of choice. `COUNT(*)` in MySQL, for example, can either be almost-free or painfully slow depending on which storage engine you're using. Other things to consider: under what conditions does your database invalidate caches, when does it sort on disk rather than in memory, when does it need to create temporary tables, etc. The final 5%, where few ever need venture, is where Mr. Silverton spends most of his time. Never once in my life have I used `SQL_SMALL_RESULT`.

2. Good problems, bad solutions

There are cases when Mr. Silverton does note a good problem. MySQL will indeed use a dynamic row format if it contains variable length fields like `TEXT` or `BLOB`, which, in this case, means sorting needs to be done on disk. The solution is not to eschew these datatypes, but rather to split off such fields into an associated table. The following schema represents this idea:

```
1. CREATE TABLE posts (  
2.     id int UNSIGNED NOT NULL AUTO_INCREMENT,  
3.     author_id int UNSIGNED NOT NULL,
```

```

4.     created timestamp NOT NULL,
5.     PRIMARY KEY(id)
6. );
7.
8. CREATE TABLE posts_data (
9.     post_id int UNSIGNED NOT NULL.
10.     body text,
11.     PRIMARY KEY(post_id)
12. );

```

3. That's just...yeah

Some of his suggestions are just mind-boggling, e.g., "remove unnecessary parentheses." It really doesn't matter whether you do `SELECT * FROM posts WHERE (author_id = 5 AND published = 1)` or `SELECT * FROM posts WHERE author_id = 5 AND published = 1`. None. Any decent DBMS is going to optimize these away. This level of detail is akin to wondering when writing a C program whether the post-increment or pre-increment operator is faster. Really, if that's where you're spending your energy, it's a surprise you've written any code at all

My list

Let's see if I fare any better. I'm going to start from the most general.

4. Benchmark, benchmark, benchmark!

You're going to need numbers if you want to make a good decision. What queries are the worst? Where are the bottlenecks? Under what circumstances am I generating bad queries? Benchmarking is will let you simulate high-stress situations and, with the aid of profiling tools, expose the cracks in your database configuration. Tools of the trade include [supersmack](#), [ab](#), and [SysBench](#). These tools either hit your database directly (e.g., supersmack) or simulate web traffic (e.g., ab).

5. Profile, profile, profile!

So, you're able to generate high-stress situations, but now you need to find the cracks. This is what profiling is for. Profiling enables you to find the bottlenecks in your configuration, whether they be in memory, CPU, network, disk I/O, or, what is more likely, some combination of all of them.

The very first thing you should do is turn on the [MySQL slow query log](#) and install [mtop](#). This will give you access to information about the

absolute worst offenders. Have a ten-second query ruining your web application? These guys will show you the query right off.

After you've identified the slow queries you should learn about the MySQL internal tools, like [EXPLAIN](#), [SHOW STATUS](#), and [SHOW PROCESSLIST](#). These will tell you what resources are being spent where, and what side effects your queries are having, e.g., whether your heinous triple-join subselect query is sorting in memory or on disk. Of course, you should also be using your usual array of command-line profiling tools like `top`, `procinfo`, `vmstat`, etc. to get more general system performance information.

6. Tighten Up Your Schema

Before you even start writing queries you have to design a schema. Remember that the memory requirements for a table are going to be around $\#entries * size\ of\ a\ row$. Unless you expect every person on the planet to register 2.8 trillion times on your website you do not in fact need to make your `user_id` column a `BIGINT`. Likewise, if a text field will always be a fixed length (e.g., a US zipcode, which always has a canonical representation of the form "XXXX-XXXX") then a `VARCHAR` declaration just adds a superfluous byte for every row.

Some people poo-poo database normalization, saying it produces unnecessarily complex schema. However, proper normalization results in a minimization of redundant data. Fundamentally that means a smaller overall footprint at the cost of performance — the usual performance/memory tradeoff found everywhere in computer science. The best approach, IMO, is to normalize first and denormalize where performance demands it. Your schema will be more logical and you won't be optimizing prematurely.

7. Partition Your Tables

Often you have a table in which only a few columns are accessed frequently. On a blog, for example, one might display entry titles in many places (e.g., a list of recent posts) but only ever display teasers or the full post bodies once on a given page. ~~Horizontal~~ vertical partitioning helps:

```
1. CREATE TABLE posts (  
2.     id int UNSIGNED NOT NULL AUTO_INCREMENT,  
3.     author_id int UNSIGNED NOT NULL,  
4.     title varchar(128),  
5.     created timestamp NOT NULL,  
6.     PRIMARY KEY(id)  
7. );
```

```

8.
9. CREATE TABLE posts_data (
10.     post_id int UNSIGNED NOT NULL,
11.     teaser text,
12.     body text,
13.     PRIMARY KEY(post_id)
14. );

```

The above represents a situation where one is optimizing for reading. Frequently accessed data is kept in one table while infrequently accessed data is kept in another. Since the data is now partitioned the infrequently access data takes up less memory. You can also optimize for writing: frequently *changed* data can be kept in one table, while infrequently changed data can be kept in another. This allows more efficient caching since MySQL no longer needs to expire the cache for data which probably hasn't changed.

8. Don't Overuse Artificial Primary Keys

Artificial primary keys are nice because they can make the schema less volatile. If we stored geography information in the US based on zip code, say, and the zip code system suddenly changed we'd be in a bit of trouble. On the other hand, many times there are perfectly fine natural keys. One example would be a join table for many-to-many relationships. What not to do:

```

1. CREATE TABLE posts_tags (
2.     relation_id int UNSIGNED NOT NULL
   AUTO_INCREMENT,
3.     post_id int UNSIGNED NOT NULL,
4.     tag_id int UNSIGNED NOT NULL,
5.     PRIMARY KEY(relation_id),
6.     UNIQUE INDEX(post_id, tag_id)
7. );

```

Not only is the artificial key entirely redundant given the column constraints, but the number of post-tag relations are now limited by the system-size of an integer. Instead one should do:

```

8. CREATE TABLE posts_tags (
9.     post_id int UNSIGNED NOT NULL,
10.     tag_id int UNSIGNED NOT NULL,
11.     PRIMARY KEY(post_id, tag_id)
12. );

```

9. Learn Your Indices

Often your choice of indices will make or break your database. For those who haven't progressed this far in their database studies, an index is a sort of hash. If we issue the query `SELECT * FROM users WHERE last_name = 'Goldstein'` and `last_name` has no index then your DBMS must scan every row of the table and compare it to the string 'Goldstein.' An index is usually a B-tree (though there are other options) which speeds up this comparison considerably.

You should probably create indices for any field on which you are selecting, grouping, ordering, or joining. Obviously each index requires space proportional to the number of rows in your table, so too many indices winds up taking more memory. You also incur a performance hit on write operations, since every write now requires that the corresponding index be updated. There is a balance point which you can uncover by profiling your code. This varies from system to system and implementation to implementation.

10. SQL is Not C

C is the canonical procedural programming language and the greatest pitfall for a programmer looking to show off his database-fu is that he fails to realize that SQL is not procedural (nor is it functional or object-oriented, for that matter). Rather than thinking in terms of data and operations on data one must think of sets of data and relationships among those sets. This usually crops up with the improper use of a subquery:

```
1. SELECT a.id,  
2.     (SELECT MAX(created)  
3.     FROM posts  
4.     WHERE author_id = a.id)  
5. AS latest_post  
6. FROM authors a
```

Since this subquery is correlated, i.e., references a table in the outer query, one should convert the subquery to a join.

```
7. SELECT a.id, MAX(p.created) AS latest_post  
8. FROM authors a  
9. INNER JOIN posts p  
10.      ON (a.id = p.author_id)  
11. GROUP BY a.id
```

11. Understand your engines

MySQL has two primary storage engines: MyISAM and InnoDB. Each has its own performance characteristics and considerations. In the broadest sense MyISAM is good for read-heavy data and InnoDB is good for write-heavy data, though there are cases where the opposite is true. The biggest gotcha is how the two differ with respect to the COUNT function.

MyISAM keeps an internal cache of table meta-data like the number of rows. This means that, generally, COUNT(*) incurs no additional cost for a well-structured query. InnoDB, however, has no such cache. For a concrete example, let's say we're trying to paginate a query. If you have a query SELECT * FROM users LIMIT 5, 10, let's say, running SELECT COUNT(*) FROM users LIMIT 5, 10 is essentially free with MyISAM but takes the same amount of time as the first query with InnoDB. MySQL has a SQL_CALC_FOUND_ROWS option which tells InnoDB to calculate the number of rows as it runs the query, which can then be retrieved by executing SELECT FOUND_ROWS(). This is very MySQL-specific, but can be necessary in certain situations, particularly if you use InnoDB for its other features (e.g., row-level locking, stored procedures, etc.).

12. MySQL specific shortcuts

MySQL provides many extensions to SQL which help performance in many common use scenarios. Among these are [INSERT ... SELECT](#), [INSERT ... ON DUPLICATE KEY UPDATE](#), and [REPLACE](#).

I rarely hesitate to use the above since they are so convenient and provide real performance benefits in many situations. MySQL has other keywords which are more dangerous, however, and should be used sparingly. These include [INSERT DELAYED](#), which tells MySQL that it is not important to insert the data immediately (say, e.g., in a logging situation). The problem with this is that under high load situations the insert might be delayed indefinitely, causing the insert queue to balloon. You can also give MySQL [index hints](#) about which indices to use. MySQL gets it right most of the time and when it doesn't it is usually because of a bad scheme or poorly written query.

13. And one for the road...

Last, but not least, read Peter Zaitsev's [MySQL Performance Blog](#) if you're into the nitty-gritty of MySQL performance. He covers many of the finer aspects of database administration and performance.

Library

This is a collection of Slides, presentations and videos on topics related to designing of high throughput, scalable, highly available websites I've been collecting for a while.

8/28/2007 Blog	<u>Inside Myspace</u>
8/28/2007 FAQ	<u>Good Memcached FAQ</u>
8/28/2007 Blog	<u>Measuring scalability</u>
8/28/2007 Blog	<u>Distributed caching with memcached</u>
8/28/2007 Blog	<u>Mailinator stats (all on a single server)</u>
8/28/2007 Blog	<u>Architecture of Mailinator</u>
8/25/2007 Slides 74	<u>Building Scalable web architectures</u>
8/25/2007 Slides 42	<u>Typepad Architecture change: Change Your Car's Tires at 100 mph</u>
8/25/2007 Slides	<u>Skype protocol (also talks about p2p connections which is critical for its scalability)</u>
8/20/2007 slides	<u>Slashdot's History of scaling Mysql</u>
8/19/2007 Slides 20	<u>Big Bad Postgres SQL</u>
8/19/2007 Slides 90	<u>Scalable internet architectures</u>
8/19/2007 Slides 59	<u>Production troubleshooting (not related to scalability... but shit happens everywhere)</u>
8/19/2007 Slides 31	<u>Clustered Logging with mod_log_spread</u>
8/19/2007 Slides 86	<u>Understanding and Building HA/LB clusters</u>
8/12/2007 Blog	<u>Multi-Master Mysql Replication</u>
8/12/2007 Blog	<u>Large-Scale Methodologies for the World Wide Web</u>
8/12/2007 Blog	<u>Scaling gracefully</u>

8/12/2007	Blog		<u>Implementing Tag cloud - The nasty way</u>
8/12/2007	Blog		<u>Normalized Data is for sissies</u>
8/12/2007	Slides		<u>APC at facebook</u>
8/6/2007	Video		<u>Plenty Of fish interview with its CEO</u>
8/6/2007	Slides		<u>PHP scalability myth</u>
8/6/2007	Slides	79	<u>High performance PHP</u>
8/6/2007	Blog		<u>Digg: PHP' s scalability and Performance</u>
8/2/2007	Blog		<u>Getting Started with Drupal</u>
8/2/2007	Blog		<u>4 Problems with Drupal</u>
8/2/2007	Video	55m	<u>Seattle Conference on Scalability: MapReduce Used on Large Data Sets</u>
8/2/2007	Video	60m	<u>Seattle Conference on Scalability: Scaling Google for Every User</u>
8/2/2007	Video	53m	<u>Seattle Conference on Scalability: VeriSign' s Global DNS Infrastructure</u>
8/2/2007	Video	53m	<u>Seattle Conference on Scalability: YouTube Scalability</u>
8/2/2007	Video	59m	<u>Seattle Conference on Scalability: Abstractions for Handling Large Datasets</u>
8/2/2007	Video	55m	<u>Seattle Conference on Scalability: Building a Scalable Resource Management</u>
8/2/2007	Video	44m	<u>Seattle Conference on Scalability: SCTPs Reliability and Fault Tolerance</u>
8/2/2007	Video	27m	<u>Seattle Conference on Scalability: Lessons In</u>

Building Scalable Systems

8/2/2007	Video	41m	<u>Seattle Conference on Scalability: Scalable Test Selection Using Source Code</u>
8/2/2007	Video	53m	<u>Seattle Conference on Scalability: Lustre File System</u>
8/2/2007	Slides	16	<u>Technology at Digg.com</u>
8/2/2007	Blog		<u>Extreme Makeover: Database or MySQL@YouTube</u>
8/2/2007	Slides	60	<u>“Real Time Mysql at Google</u>
8/2/2007	Blog		<u>Scaling Twitter</u>
8/2/2007	Slides	56	<u>How we build Vox</u>
8/2/2007	Slides	97	<u>High Performance websites</u>
8/2/2007	Slides	101	<u>Beyond the file system design</u>
8/2/2007	Slides	145	<u>Scalable web architectures</u>
8/2/2007	Blog		<u>“Build Scalable Web 2.0 Sites with Ubuntu</u>
8/2/2007	Slides	34	<u>Scalability set Amazon’s servers on fire not yours</u>
8/2/2007	Slides	41	<u>Hardware layouts for LAMP installations</u>
8/2/2007	Video	91m	<u>Mysql scaling and high availability architectures</u>
8/2/2007	Audio	137	<u>Lessons from Building world’s largest social music platform</u>
8/2/2007	PDF	137	<u>Lessons from Building world’s largest social music platform</u>
8/2/2007	Slides	137	<u>Lessons from Building world’s largest social music platform</u>

8/2/2007	PDF	80	<u>Livejournal's backend: history of scaling</u>
8/2/2007	Slides	80	<u>Livejournal's backend: history of scaling</u>
8/2/2007	Slides	26	<u>Scalable Web Architectures (w/ Ruby and Amazon S3)</u>
8/2/2007	Blog		<u>Yahoo! bookmarks uses symfony</u>
8/2/2007	Slides		<u>Getting Rich with PHP 5</u>
8/2/2007	Audio		<u>Getting Rich with PHP 5</u>
8/2/2007	Blog		<u>Scaling Fast and Cheap - How We Built Flickr</u>
8/2/2007	News		<u>Open source helps Flickr share photos</u>
8/2/2007	Slides	41	<u>Flickr and PHP</u>
8/2/2007	Slides	30	<u>Wikipedia: Cheap and explosive scaling with LAMP</u>
8/2/2007	Blog		<u>YouTube Scalability Talk</u>
8/2/2007			<u>High Order Bit: Architecture for Humanity</u>
8/2/2007	PDF		<u>Mysql and Web2.0 companies</u>
8/3/2007		36	<u>Building Highly Scalable Web Applications</u>
8/3/2007			<u>Introduction to hadoop</u>
8/3/2007	webpage		<u>The Hadoop Distributed File System: Architecture and Design</u>
8/3/2007			<u>Interpreting the Data: Parallel Analysis with Sawzall</u>
8/3/2007	PDF		<u>ODISSEA: A Peer-to-Peer Architecture for Scalable Web Search and Information Retrieval</u>
8/3/2007	PDF		<u>SEDA: An Architecture for well conditioned scalable internet services</u>

8/3/2007	PDF	<u>A scalable architecture for Global web service hosting service</u>
8/3/2007		<u>Meed Hadoop</u>
8/3/2007	Blog	<u>Yahoo's Hadoop Support</u>
8/3/2007	Blog	<u>Running Hadoop MapReduce on Amazon EC2 and Amazon S3</u>
8/3/2007	53m	<u>LH*RSP2P : A Scalable Distributed Data Structure for P2P Environment</u>
8/3/2007	90m	<u>Scaling the Internet routing table with Locator/ID Separation Protocol (LISP)</u>
8/3/2007		<u>Hadoop Map/Reduce</u>
8/3/2007	Slides	<u>Hadoop distributed file system</u>
8/3/2007	Video 45m	<u>Brad Fitzpatrick - Behind the Scenes at LiveJournal : Scaling Storytime</u>
8/3/2007	Slides	<u>Inside LiveJournal's Backend (April 2004)</u>
8/3/2007	Slides 25	<u>How to scale</u>
8/3/2007	36m	<u>Testing Oracle 10g RAC Scalability</u>
8/3/2007	Slides 107	<u>PHP & Performance</u>
8/3/2007	Blog 1217	
8/3/2007	45m	<u>SQL Performance Optimization</u>
8/3/2007	80m	<u>Building a Scalable Software Security Practice</u>
8/3/2007	59m	<u>Building Large Systems at Google</u>
8/3/2007		<u>Scalable computing with Hadoop</u>

8/3/2007	Slides	37	<u>The Ebay architecture</u>
8/3/2007	PDF		<u>Bigtable: A Distributed Storage System for Structured Data</u>
8/3/2007	PDF		<u>Fault-Tolerant and scalable TCP splice and web server architecture</u>
8/3/2007	Video		<u>BigTable: A Distributed Structured Storage System</u>
8/3/2007	PDF		<u>MapReduce: Simplified Data Processing on Large Clusters</u>
8/3/2007	PDF		<u>Google Cluster architecture</u>
8/3/2007	PDF		<u>Google File System</u>
8/3/2007	Doc		<u>Implementing a Scalable Architecture</u>
8/3/2007	News		<u>How linux saved Millions for Amazon</u>
8/3/2007			<u>Yahoo experience with hadoop</u>
8/3/2007	Slides		<u>Scalable web application using Mysql and Java</u>
8/3/2007	Slides		<u>Friendster: scaling for 1 Billion Queries per day</u>
8/3/2007	Blog		<u>Lightweight web servers</u>
8/3/2007	PDF		<u>Mysql Scale out by application partitioning</u>
8/3/2007	PDF		<u>Replication under scalable hashing: A family of algorithms for Scalable decentralized data distribution</u>
8/3/2007	Product		<u>Clustered storage revolution</u>
8/3/2007	Blog		<u>Early Amazon Series</u>
8/3/2007	Web		<u>Wikimedia Server info</u>

8/3/2007	Slides	32	Wikimedia Architecture
8/3/2007	Slides	21	MySpace presentation
8/3/2007	PDF		A scalable and fault-tolerant architecture for distributed web resource discovery
8/4/2007	PDF		The Chubby Lock Service for Loosely-Coupled Distributed Systems
8/5/2007	Slides	47	Real world Mysql tuning
8/5/2007	Slides	100	Real world Mysql performance tuning
8/5/2007	Slides	63	Learning MogileFS: Building scalable storage system
8/5/2007	Slides		Reverse Proxy and Webserver
8/5/2007	PDF		Case for Shared Nothing
8/5/2007	Slides	27	A scalable stateless proxy for DBI
8/5/2007	Slides	91	Real world scalability web builder 2006
8/5/2007	Slides	52	Real world web scalability

Friendster Architecture



Thu, 07/12/2007 - 05:18 — Todd Hoff

- **Friendster Architecture (341)**

Friendster is one of the largest social network sites on the web. it emphasizes genuine friendships and the discovery of new people through friends.

Site: <http://www.friendster.com/>

Information Sources

- **Friendster** - Scaling for 1 Billion Queries per day

Platform

- **MySQL**
- **Perl**
- **PHP**
- **Linux**
- **Apache**

What's Inside?

- Dual x86-64 AMD Opterons with 8 GB of RAM
- Faster disk (**SAN**)
- Optimized indexes
- Traditional 3-tier architecture with hardware load balancer in front of the databases
- Clusters based on types: ad, app, photo, monitoring, DNS, gallery search DB, profile DB, user infor DB, IM status cache, message DB, testimonial DB, friend DB, graph servers, gallery search, object cache.

Lessons Learned

- No persistent database connections.
- Removed all sorts.
- Optimized indexes
- Don't go after the biggest problems first
- Optimize without downtime
- Split load
- Moved sorting query types into the application and added LIMITS.
- Reduced ranges

- Range on primary key
- Benchmark -> Make Change -> Benchmark -> Make Change (Cycle of Improvement)
- Stabilize: always have a plan to rollback
- Work with a team
- Assess: Define the issues
- A key design goal for the new system was to move away from maintaining session state toward a stateless architecture that would clean up after each request
- Rather than buy big, centralized boxes, [our philosophy] was about buying a lot of thin, cheap boxes. If one fails, you roll over to another box.

I Feedblendr Architecture - Using EC2 to Scale



Wed, 10/31/2007 - 05:15 — [Todd Hoff](#)

- [Feedblendr Architecture - Using EC2 to Scale \(56\)](#)

A man had a dream. His dream was to blend a bunch of RSS/Atom/RDF feeds into a single feed. The man is Beau Lebens of [Feedville](#) and like most dreamers he was a little short on coin. So he took refuge in the home of a cheap hosting provider and Beau realized his dream, creating [FEEDblendr](#). But FEEDblendr chewed up so much CPU creating blended feeds that the cheap hosting provider ordered Beau to find another home. Where was Beau to go? He eventually found a new home in the virtual machine room of Amazon's [EC2](#). This is the story of how Beau was finally able to create his one feeds safe within the cradle of affordable CPU cycles.

Site: <http://feedblendr.com/>

The Platform

- EC2 (Fedora Core 6 Lite distro)
- S3
- Apache
- PHP
- MySQL
- DynDNS (for round robin DNS)

The Stats

- Beau is a developer with some sysadmin skills, not a web server admin, so a lot of learning was involved in creating FEEDblendr.
- FEEDblendr uses 2 EC2 instances. The same Amazon Instance (AMI) is used for both instances.
- Over 10,000 blends have been created, containing over 45,000 source feeds.
- Approx 30 blends created per day. Processors on the 2 instances are actually pegged pretty high (load averages at ~ 10 - 20 most of the time).

The Architecture

- Round robin DNS is used to load balance between instances.

-The DNS is updated by hand as an instance is validated to work correctly before the DNS is updated.

-Instances seem to be more stable now than they were in the past, but you must still assume they can be lost at any time and no data will be persisted between reboots.

- The database is still hosted on an external service because EC2 does not have a decent persistent storage system.

- The AMI is kept as minimal as possible. It is a clean instance with some auto-deployment code to load the application off of S3. This means you don't have to create new instances for every software release.
- The deployment process is:
 - Software is developed on a laptop and stored in subversion.
 - A makefile is used to get a revision, fix permissions etc, package and push to S3.
 - When the AMI launches it runs a script to grab the software package from S3.
 - The package is unpacked and a specific script inside is executed to continue the installation process.
 - Configuration files for Apache, PHP, etc are updated.
 - Server-specific permissions, symlinks etc are fixed up.
 - Apache is restarted and email is sent with the IP of that machine. Then the DNS is updated by hand with the new IP address.
- Feeds are intelligently cached independently on each instance. This is to reduce the costly polling for feeds as much as possible. S3 was tried as a common feed cache for both instances, but it was too slow. Perhaps feeds could be written to each instance so they would be cached on each machine?

Lesson Learned

- A low budget startup can effectively bootstrap using EC2 and S3.
- For the budget conscious the free ZoneEdit service might work just as well as the \$50/year DynDNS service (which works fine).
- Round robin load balancing is slow and unreliable. Even with a short TTL for the DNS some systems hold on to the IP addressed for a long time, so new machines are not load balanced to.
- Many problems exist with RSS implementations that keep feeds from being

effectively blended. A lot of CPU is spent reading and blending feeds unnecessarily because there's no reliable cross implementation way to tell when a feed has really changed or not.

- It's really a big mindset change to consider that your instances can go away at any time. You have to change your architecture and design to live with this fact. But once you internalize this model, most problems can be solved.
- EC2's poor load balancing and persistence capabilities make development and deployment a lot harder than it should be.
- Use the AMI's ability to be passed a parameter to select which configuration to load from S3. This allows you to test different configurations without moving/deleting the current active one.
- Create an automated test system to validate an instance as it boots. Then automatically update the DNS if the tests pass. This makes it easy create new instances and takes the slow human out of the loop.
- Always load software from S3. The last thing you want happening is your instance loading, and for some reason not being able to contact your SVN server, and thus failing to load properly. Putting it in S3 virtually eliminates the chances of this occurring, because it's on the same network.

Related Articles

- [What is a 'River of News' style aggregator?](#)
- [Build an Infinitely Scalable Infrastructure for \\$100 Using Amazon Services](#)
- [EC2](#)
- [Example](#)
- [MySQL](#)
- [PHP](#)

- [S3](#)
- [Visit Feedblendr Architecture - Using EC2 to Scale](#)
- 716 reads

Comments

Wed, 10/31/2007 - 15:04 — [Greg Linden](#) (not verified)

Re: Feedblendr Architecture - Using EC2 to Scale

I might be missing something, but I don't see how this is an interesting example of "using **EC2** to scale".

There appears to be no difference between using EC2 in the way Beau is using it and setting up two leased servers from a normal provider. In fact, getting leased servers might be better, since the cost might be lower (an EC2 instance costs \$72/month + bandwidth) and the database would be on the same network.

Beau does not appear to be doing anything that takes advantage of EC2, such as dynamically creating and discarding instances based on demand.

Am I missing something here? Is this an interesting use of using EC2 to scale?

- [reply](#)

Wed, 10/31/2007 - 16:35 — [Todd Hoff](#)



Re: Feedblendr Architecture - Using EC2 to Scale

> I might be missing something, but I don't see how this is an interesting example of "using **EC2** to scale".

I admit to being a bit polymorphously perverse with respect to finding things interesting, but from Beau's position, which many people are, the drama is thrilling. The story starts with a conflict: how to implement this idea? The first option is the traditional cheap host option. And for a long time that would have been the end of the story. Dedicated servers with higher end CPUs, RAM, and persistent storage are still not cheap. So if you aren't making money that would have been where the story ended. Scaling by adding more and more dedicated servers would be impossible. Hopefully the new grid model will allow a lot of people to keep writing their stories. His learning curve of creating the system is what was most interesting. Figuring out how to set things up, load balance, load the software, test it, regular nuts and bolts development stuff. And that puts him in the position of being able to get more CPU immediately when and if the time comes. He'll be able to add that feature in quickly because he's already done the ground work. But for now it's running fine. The spanner in the plan was the database and that points out the fatal flaw of EC2, which is the database. The plan would look a bit more successful if that part had worked out better, but it didn't, which is also interesting.

- [reply](#)

Wed, 10/31/2007 - 18:41 — [Beau Lebens](#) (not verified)

Re: Feedblendr Architecture - Using EC2 to Scale

@Todd, thanks for the write-up, and a couple quick corrections/clarifications:

- "Beau is a developer with some sysadmin skills, not a web server admin, so a lot of learning was involved in creating FEEDblendr." - Just to be clear, the learning curve was mostly in dealing with EC2 and how it works, not so much FeedBlendr, which at it's core is relatively simple.

- "no data will be persisted between reboots" this is not exactly true. Rebooting will persist data, but a true "crash" or termination of your instance will discard everything.

- "The database is still hosted on an external service because EC2 does not have a decent persistent storage system" - more the case here is that I didn't want to have to deal with (or pay for) setting something up to cater to them not having persistent storage. It is being done by other people, and can be done, it just seemed like overkill for what I was doing.

- "EC2's poor load balancing and persistence capabilities make development and **deployment** a lot harder than it should be" - to be clear, EC2 has no inherent load balancing, so it's up to you (the developer/admin) to provide it yourself somehow. There are a number of different ways of doing it, but I choose dynamic DNS because it was something I was familiar with.

@Greg in response to your question - I suppose the point here is that even though FeedBlendr isn't currently a poster-child for scaling, that's also kind of the point. As Todd says, this is about the learning curve and trials and tribulations of getting to a point where it can scale. There is nothing stopping me (other than budget!) from launching an additional 5 instances right now and adding them into DNS, and then I've suddenly scaled. From there I can kill some instances off and scale back. This is all about getting to the point where I even have that option, and how it was done on EC2 in particular.

Cheers,

Beau

I PlentyOfFish Architecture



Tue, 10/30/2007 - 04:48 — [Todd Hoff](#)

- [PlentyOfFish Architecture \(983\)](#)

*Update: by Facebook standards Read/WriteWeb says POF is worth a cool **one billion dollars**.* It helps to talk like Dr. Evil when saying it out loud.

PlentyOfFish is a hugely popular on-line dating system slammed by over 45 million visitors a month and 30+ million hits a day (500 - 600 pages per second). But that's not the most interesting part of the story. All this is handled by one person, using a handful of servers, working a few hours a day, while making \$6 million a year from Google ads. Jealous? I know I am. How are all these love connections made using so few resources?

Site: <http://www.plentyoffish.com/>

Information Sources

- [Channel9 Interview with Markus Frind](#)
- [Blog of Markus Frind](#)
- [Plentyoffish: 1-Man Company May Be Worth \\$1Billion](#)

The Platform

- Microsoft Windows
- **ASP.NET**
- **IIS**
- Akamai **CDN**
- Foundry ServerIron Load Balancer

The Stats

- PlentyOfFish (POF) gets 1.2 billion page views/month, and 500,000 average unique logins per day. The peak season is January, when it will grow 30 percent.
- POF has one single employee: the founder and CEO Markus Frind.

- Makes up to \$10 million a year on Google ads working only two hours a day.
- 30+ Million Hits a Day (500 - 600 pages per second).
- 1.1 billion page views and 45 million visitors a month.
- Has 5-10 times the click through rate of Facebook.
- A top 30 site in the US based on Competes Attention metric, top 10 in Canada and top 30 in the UK.
- 2 load balanced web servers with 2 Quad Core Intel Xeon X5355 @ 2.66Ghz), 8 Gigs of RAM (using about 800 MBs), 2 hard drives, runs Windows x64 Server 2003.
- 3 DB servers. No data on their configuration.
- Approaching 64,000 simultaneous connections and 2 million page views per hour.
- Internet connection is a 1Gbps line of which 200Mbps is used.
- 1 TB/day serving 171 million images through Akamai.
- 6TB storage array to handle millions of full sized images being uploaded every month to the site.

What's Inside

- Revenue model has been to use Google ads. Match.com, in comparison, generates \$300 million a year, primarily from subscriptions. POF's revenue model is about to change so it can capture more revenue from all those users. The plan is to hire more employees, hire sales people, and sell ads directly instead of relying solely on AdSense.
- With 30 million page views a day you can make good money on advertising, even a 5 - 10 cents a CPM.
- Akamai is used to serve 100 million plus image requests a day. If you have 8 images and each takes 100 msecs you are talking a second load just for the images. So distributing the images makes sense.
- 10's of millions of image requests are served directly from their servers, but the

majority of these images are less than 2KB and are mostly cached in RAM.

- Everything is dynamic. Nothing is static.
- All outbound Data is Gzipped at a cost of only 30% CPU usage. This implies a lot of processing power on those servers, but it really cuts bandwidth usage.
- No caching functionality in ASP.NET is used. It is not used because as soon as the data is put in the cache it's already expired.
- No built in components from ASP are used. Everything is written from scratch.

Nothing is more complex than a simple if then and for loops. Keep it simple.

- Load balancing

- IIS arbitrarily limits the total connections to 64,000 so a load balancer was added to handle the large number of simultaneous connections. Adding a second IP address and then using a round robin DNS was considered, but the load balancer was considered more redundant and allowed easier swap in of more web servers. And using ServerIron allowed advanced functionality like bot blocking and load balancing based on passed on cookies, session data, and IP data.

- The Windows Network **Load Balancing** (NLB) feature was not used because it doesn't do sticky sessions. A way around this would be to store session state in a database or in a shared file system.

- 8-12 NLB servers can be put in a farm and there can be an unlimited number of farms. A DNS round-robin scheme can be used between farms. Such an architecture has been used to enable 70 front end web servers to support over 300,000 concurrent users.

- NLB has an affinity option so a user always maps to a certain server, thus no external storage is used for session state and if the server fails the user loses their state and must relogin. If this state includes a shopping cart or other important data, this solution may be poor, but for a dating site it seems reasonable.

- It was thought that the cost of storing and fetching session data in software was too

expensive. Hardware load balancing is simpler. Just map users to specific servers and if a server fails have the user log in again.

- The cost of a ServerIron was cheaper and simpler than using NLB. Many major sites use them for TCP connection pooling, automated bot detection, etc. ServerIron can do a lot more than load balancing and these features are attractive for the cost.

- Has a big problem picking an ad server. Ad server firms want several hundred thousand a year plus they want multi-year contracts.

- In the process of getting rid of ASP.NET repeaters and instead uses the append string thing or response.write. If you are doing over a million page views a day just write out the code to spit it out to the screen.

- Most of the build out costs went towards a **SAN**. Redundancy at any cost.

- Growth was through word of mouth. Went nuts in Canada, spread to UK, Australia, and then to the US.

- **Database**

- One database is the main database.

- Two databases are for search. Load balanced between search servers based on the type of search performed.

- Monitors performance using task manager. When spikes show up he investigates.

Problems were usually blocking in the database. It's always database issues. Rarely any problems in .net. Because POF doesn't use the .net library it's relatively easy to track down performance problems. When you are using many layers of frameworks finding out where problems are hiding is frustrating and hard.

- If you call the database 20 times per page view you are screwed no matter what you do.

- Separate database reads from writes. If you don't have a lot of RAM and you do reads and writes you get paging involved which can hang your system for seconds.

- Try and make a read only database if you can.

- Denormalize data. If you have to fetch stuff from 20 different tables try and make one table that is just used for reading.
- One day it will work, but when your database doubles in size it won't work anymore.
- If you only do one thing in a system it will do it really really well. Just do writes and that's good. Just do reads and that's good. Mix them up and it messes things up. You run into locking and blocking issues.
- If you are maxing the CPU you've either done something wrong or it's really really optimized. If you can fit the database in RAM do it.
- The development process is: come up with an idea. Throw it up within 24 hours. It kind of half works. See what user response is by looking at what they actually do on the site. Do messages per user increase? Do session times increase? If people don't like it then take it down.
- System failures are rare and short lived. Biggest issues are DNS issues where some ISP says POF doesn't exist anymore. But because the site is free, people accept a little down time. People often don't notice sites down because they think it's their problem.
- Going from one million to 12 million users was a big jump. He could scale to 60 million users with two web servers.
- Will often look at competitors for ideas for new features.
- Will consider something like S3 when it becomes geographically load balanced.

Lessons Learned

- You don't need millions in funding, a sprawling infrastructure, and a building full of employees to create a world class website that handles a torrent of users while making good money. All you need is an idea that appeals to a lot of people, a site that takes off by word of mouth, and the experience and vision to build a site without falling into the typical traps of the trade. That's all you need :-)

- Necessity is the mother of all change.
- When you grow quickly, but not too quickly you have a chance grow, modify, and adapt.
- RAM solves all problems. After that it's just growing using bigger machines.
- When starting out keep everything as simple as possible. Nearly everyone gives this same advice and Markus makes a noticeable point of saying everything he does is just obvious common sense. But clearly what is simple isn't merely common sense. Creating simple things is the result of years of practical experience.
- Keep database access fast and you have no issues.
- A big reason POF can get away with so few people and so little equipment is they use a CDN for serving large heavily used content. Using a CDN may be the secret sauce in a lot of large websites. Markus thinks there isn't a single site in the top 100 that doesn't use a CDN. Without a CDN he thinks load time in Australia would go to 3 or 4 seconds because of all the images.
- Advertising on Facebook yielded poor results. With 2000 clicks only 1 signed up. With a CTR of 0.04% Facebook gets 0.4 clicks per 1000 ad impressions, or .4 clicks per CPM. At 5 cent/CPM = 12.5 cents a click, 50 cent/CPM = \$1.25 a click. \$1.00/CPM = \$2.50 a click. \$15.00/CPM = \$37.50 a click.
- It's easy to sell a few million page views at high CPM's. It's a LOT harder to sell billions of page views at high CPM's, as shown by Myspace and Facebook.
- The ad-supported model limits your revenues. You have to go to a paid model to grow larger. To generate 100 million a year as a free site is virtually impossible as you need too big a market.
- Growing page views via Facebook for a dating site won't work. Having a visitor on you site is much more profitable. Most of Facebook's page views are outside the US and you have to split 5 cent CPM's with Facebook.

- Co-req is a potential large source of income. This is where you offer in your site's sign up to send the user more information about mortgages are some other product.
- You can't always listen to user responses. Some users will always love new features and others will hate it. Only a fraction will complain. Instead, look at what features people are actually using by watching your site.

I Wikimedia architecture



Wed, 08/22/2007 - 23:56 — Todd Hoff

- [Wikimedia architecture \(566\)](#)

Wikimedia is the platform on which Wikipedia, Wiktionary, and the other seven wiki dwarfs are built on. This document is just excellent for the student trying to scale the heights of giant websites. It is full of details and innovative ideas that have been proven on some of the most used websites on the internet.

Site: <http://wikimedia.org/>

Information Sources

- [Wikimedia architecture](#)
- http://meta.wikimedia.org/wiki/Wikimedia_servers
- [scale-out vs scale-up](#) in the from [Oracle](#) to [MySQL](#) blog.

Platform

- [Apache](#)
- [Linux](#)

- MySQL
- PHP
- Squid
- LVS
- Lucene for Search
- Memcached for Distributed Object Cache
- Lighttpd Image Server

The Stats

- 8 million articles spread over hundreds of language projects (english, dutch, ...)
- 10th busiest site in the world (source: Alexa)
- Exponential growth: doubling every 4-6 months in terms of visitors / traffic / servers
- 30 000 HTTP requests/s during peak-time
- 3 Gbit/s of data traffic
- 3 data centers: Tampa, Amsterdam, Seoul
- 350 servers, ranging between 1x P4 to 2x Xeon Quad-Core, 0.5 - 16 GB of memory
- managed by ~ 6 people
- 3 clusters on 3 different continents

The Architecture

- Geographic **Load Balancing**, based on source IP of client resolver, directs clients to the nearest server cluster. Statically mapping IP addresses to countries to clusters
- HTTP reverse proxy caching implemented using Squid, grouped by text for wiki content and media for images and large static files.
- 55 Squid servers currently, plus 20 waiting for setup.
- 1,000 HTTP requests/s per server, up to 2,500 under stress

- ~ 100 - 250 Mbit/s per server
- ~ 14 000 - 32 000 open connections per server
- Up to 40 GB of disk caches per Squid server
- Up to 4 disks per server (1U rack servers)
- 8 GB of memory, half of that used by Squid
- Hit rates: 85% for Text, 98% for Media, since the use of **CARP**.
- **PowerDNS** provides geographical distribution.
- In their primary and regional data center they build text and media clusters built on LVS, CARP Squid, Cache Squid. In the primary datacenter they have the media storage.
- To make sure the latest revision of all pages are served invalidation requests are sent to all Squid caches.
- One centrally managed & synchronized software installation for hundreds of wikis.
- MediaWiki scales well with multiple CPUs, so we buy dual quad-core servers now (8 CPU cores per box)
- Hardware shared with External Storage and Memcached tasks
- Memcached is used to cache image metadata, parser data, differences, users and sessions, and revision text. Metadata, such as article revision history, article relations (links, categories etc.), user accounts and settings are stored in the core databases
- Actual revision text is stored as blobs in External storage
- Static (uploaded) files, such as images, are stored separately on the image server - metadata (size, type, etc.) is cached in the core database and object caches
- Separate database per wiki (not separate server!)
- One master, many replicated slaves
- Read operations are load balanced over the slaves, write operations go to the master
- The master is used for some read operations in case the slaves are not yet up to date (lagged)

- External Storage
 - Article text is stored on separate data storage clusters, simple append-only blob storage. Saves space on expensive and busy core databases for largely unused data
 - Allows use of spare resources on application servers (2x 250-500 GB per server)
 - Currently replicated clusters of 3 MySQL hosts are used; this might change in the future for better manageability

Lessons Learned

- Focus on architecture, not so much on operations or nontechnical stuff.
- Sometimes caching costs more than recalculating or looking up at the data source...profiling!
- Avoid expensive algorithms, database queries, etc.
- Cache every result that is expensive and has temporal locality of reference.
- Focus on the hot spots in the code (profiling!).
- Scale by separating:
 - Read and write operations (master/slave)
 - Expensive operations from cheap and more frequent operations (query groups)
 - Big, popular wikis from smaller wikis
- Improve caching: temporal and spatial locality of reference and reduces the data set size per server
- Text is compressed and only revisions between articles are stored.
- Simple seeming library calls like using stat to check for a file's existence can take too long when loaded.
- Disk seek I/O limited, the more disk spindles, the better!
- Scale-out using commodity hardware doesn't require using cheap hardware.

Wikipedia's database servers these days are 16GB dual or quad core boxes with 6 15,000 RPM SCSI drives in a **RAID 0** setup. That happens to be the sweet spot for the working set and load balancing setup they have. They would use smaller/cheaper systems if it made sense, but 16GB is right for the working set size and that drives the rest of the spec to match the demands of a system with that much RAM. Similarly the web servers are currently 8 core boxes because that happens to work well for load balancing and gives good PHP throughput with relatively easy load balancing.

- It is a lot of work to scale out, more if you didn't design it in originally. Wikipedia's MediaWiki was originally written for a single master database server. Then slave support was added. Then partitioning by language/project was added. The designs from that time have stood the test well, though with much more refining to address new bottlenecks.
- Anyone who wants to design their database architecture so that it'll allow them to inexpensively grow from one box rank nothing to the top ten or hundred sites on the net should start out by designing it to handle slightly out of date data from replication slaves, know how to load balance to slaves for all read queries and if at all possible to design it so that chunks of data (batches of users, accounts, whatever) can go on different servers. You can do this from day one using virtualisation, proving the architecture when you're small. It's a LOT easier than doing it while load is doubling every few months!

I Scaling Early Stage Startups



Mon, 10/29/2007 - 04:26 — [Todd Hoff](#)

- [Scaling Early Stage Startups \(56\)](#)

Mark Maunder of [No VC Required](http://novcrequired.com/)--who advocates not taking VC money lest you be turned into a frog instead of the prince (or princess) you were dreaming of--has an excellent [slide deck](#) on how to scale an early stage startup. His blog also has some good SEO tips and a very spooky widget showing the geographical location of his readers. Perfect for Halloween! What is Mark's other worldly scaling strategies for startups?

Site: <http://novcrequired.com/>

Information Sources

- [Slides from Seattle Tech Startup Talk](#).
- [Scaling Early Stage Startups](#) blog post by Mark Maunder.

The Platform

- Linux
- An ISAM type data store.
- [Perl](#)
- [Httpperf](#) is used for benchmarking.
- [Websitepulse.com](#) is used for perf monitoring.

The Architecture

- [Performance](#) matters because being slow could cost you 20% of your revenue. The UIE guys disagree saying this ain't necessarily so. They explain their reasoning in [Usability Tools Podcast: The Truth About Page Download Time](#). The idea is: "There was still another surprising finding from our study: a strong correlation between perceived download time and whether users successfully completed their tasks on a site. There was, however, no correlation between actual download time and task success, causing

us to discard our original hypothesis. It seems that, when people accomplish what they set out to do on a site, they perceive that site to be fast." So it might be a better use of time to improve the front-end rather than the back-end.

- **MySQL** was dumped because of performance problems: MySQL didn't handle a high number of writes and deletes on large tables, writes blow away the query cache, large numbers of small tables (over 10,000) are not well supported, uses a lot of memory to cache indexes, maxed out at 200 concurrent read/write queries per second with over 1 million records.
- For data storage they evolved to a fixed length ISAM like record scheme that allows seeking directly to the data. Still uses file level locking and its benchmarked at 20,000+ concurrent reads/writes/deletes. Considering moving to BerkelyDB which is a very highly performing and is used by many large websites, especially when you primarily need key-value type lookups. I think it might be interesting to store json if a lot of this data ends up being displayed on the web page.
- Moved to httpd.prefork for Perl. That with no keepalive on the application servers uses less RAM and works well.

Lessons Learned

- Configure your DB and web server correctly. MySQL and **Apache**'s memory usage can easily spiral out of control which leads gridingly slow performance as swapping increases. Here are a few resources for helping with [configuration issues](#).
- Serve only the users you care about. Block content thieves that crawl your site using a lot of valuable resources for nothing. Monitor the number of content pages they fetch per minute. If a threshold is exceeded and then do a reverse lookup on their IP address and configure your firewall to block them.
- Cache as much DB data and static content as possible. Perl's Cache::FileCache was

used to cache DB data and rendered HTML on disk.

- Use two different host names in URLs to enable browser clients to load images in parallel.
- Make content as static as possible Create a separate Image and CSS server to serve the static content. Use keepalives on static content as static content uses little memory per thread/process.
- Leave plenty of spare memory. Spare memory allows **Linux** to use more memory for file system caching which increased performance about 20 percent.
- Turn Keepalive off on your dynamic content. Increasing http requests can exhaust the thread and memory resources needed to serve them.
- You may not need a complex RDBMS for accessing data. Consider a lighter weight database BerkelyDB.

I Database parallelism choices greatly impact scalability

By Sam Madden on October 30, 2007 9:15 AM | [Permalink](#) | [Comments \(2\)](#) | [TrackBacks \(0\)](#)

Large databases require the use of parallel computing resources to get good performance. There are several fundamentally different parallel architectures in use today; in this post, Dave DeWitt, Mike Stonebraker, and I review three approaches and reflect on the pros and cons of each. Though these tradeoffs were articulated in the research community twenty years ago, we wanted to revisit these issues to bring readers up to speed before publishing upcoming posts that will discuss recent developments in parallel database design.

Shared-memory systems don't scale well as the shared bus becomes the bottleneck

In a shared-memory approach, as implemented on many symmetric multi-processor machines, all of the CPUs share a single memory and a single collection of disks. This approach is relatively easy to program. Complex distributed locking and commit protocols are not needed because the lock manager and buffer pool are both stored in the memory system where they can be easily accessed by all the processors.

Unfortunately, shared-memory systems have fundamental scalability limitations, as all I/O and memory requests have to be transferred over the same bus that all of the processors

share. This causes the bandwidth of the bus to rapidly become a bottleneck. In addition, shared-memory multiprocessors require complex, customized hardware to keep their L2 data caches consistent. Hence, it is unusual to see shared-memory machines of larger than 8 or 16 processors unless they are custom-built from non-commodity parts (and if they are custom-built, they are very expensive). As a result, shared-memory systems don't scale well.

Shared-disk systems don't scale well either

Shared-disk systems suffer from similar scalability limitations. In a shared-disk architecture, there are a number of independent processor nodes, each with its own memory. These nodes all access a single collection of disks, typically in the form of a storage area network (SAN) system or a network-attached storage (NAS) system. This architecture originated with the Digital Equipment Corporation VAXcluster in the early 1980s, and has been widely used by Sun Microsystems and Hewlett-Packard.

Shared-disk architectures have a number of drawbacks that severely limit scalability. First, the interconnection network that connects each of the CPUs to the shared-disk subsystem can become an I/O bottleneck. Second, since there is no pool of memory that is shared by all the processors, there is no obvious place for the lock table or buffer pool to reside. To set locks, one must either centralize the lock manager on one processor or resort to a complex distributed locking protocol. This protocol must use messages to implement in software the same sort of cache-consistency protocol implemented by shared-memory multiprocessors in hardware. Either of these approaches to locking is likely to become a bottleneck as the system is scaled.

To make shared-disk technology work better, vendors typically implement a "shared-cache" design. Shared cache works much like shared disk, except that, when a node in a parallel cluster needs to access a disk page, it first checks to see if the page is in its local buffer pool ("cache"). If not, it checks to see if the page is in the cache of any other node in the cluster. If neither of those efforts works, it reads the page from disk.

Such a cache appears to work fairly well on OLTP but performs less well for data warehousing workloads. The problem with the shared-cache design is that cache hits are unlikely to happen because warehouse queries are typically answered using sequential scans of the fact table (or via materialized views). Unless the whole fact table fits in the aggregate memory of the cluster, sequential scans do not typically benefit from large amounts of cache. Thus, the entire burden of answering such queries is placed on the disk subsystem. As a result, a shared cache just creates overhead and limits scalability.

In addition, the same scalability problems that exist in the shared memory model also occur in the shared-disk architecture. The bus between the disks and the processors will likely become a bottleneck, and resource contention for certain disk blocks, particularly as the number of CPUs increases, can be a problem. To reduce bus contention, customers

frequently configure their large clusters with many Fiber channel controllers (disk buses), but this complicates system design because now administrators must partition data across the disks attached to the different controllers.

Shared-nothing scales the best

In a shared-nothing approach, by contrast, each processor has its own set of disks. Data is "horizontally partitioned" across nodes. Each node has a subset of the rows from each table in the database. Each node is then responsible for processing only the rows on its own disks. Such architectures are especially well suited to the star schema queries present in data warehouse workloads, as only a very limited amount of communication bandwidth is required to join one or more (typically small) dimension tables with the (typically much larger) fact table.

In addition, every node maintains its own lock table and buffer pool, eliminating the need for complicated locking and software or hardware consistency mechanisms. Because shared nothing does not typically have nearly as severe bus or resource contention as shared-memory or shared-disk machines, shared nothing can be made to scale to hundreds or even thousands of machines. Because of this, it is generally regarded as the best-scaling architecture.

Approach	Shared memory	Shared disk	Shared nothing
Scalability	Low	Moderate	High
Vendors following the approach	Microsoft SQL Server MySQL PostgreSQL	Oracle RAC Sybase IQ	IBM DB2 Netezza Teradata Vertica

The shared nothing approach compliments other enhancements

As a closing point, we note that this shared nothing approach is completely compatible with other advanced database techniques we've discussed on this blog, such as compression and vertical partitioning. Systems that combine all of these techniques are likely to offer the best performance and scalability when compared to more traditional architectures.

I Introduction to Distributed System Design

Table of Contents

[Audience and Pre-Requisites](#)

[The Basics](#)

[So How Is It Done?](#)

Audience and Pre-Requisites

This tutorial covers the basics of distributed systems design. The pre-requisites are significant programming experience with a language such as C++ or Java, a basic understanding of networking, and data structures & algorithms.

The Basics

What is a distributed system? It's one of those things that's hard to define without first defining many other things. Here is a "cascading" definition of a distributed system:

A program

is the code you write.

A process

is what you get when you run it.

A message

is used to communicate between processes.

A packet

is a fragment of a message that might travel on a wire.

A protocol

is a formal description of message formats and the rules that two processes must follow in order to exchange those messages.

A network

is the infrastructure that links computers, workstations, terminals, servers, etc. It consists of routers which are connected by communication links.

A component

can be a process or any piece of hardware required to run a process, support communications between processes, store data, etc.

A distributed system

is an application that executes a collection of protocols to coordinate the actions of multiple processes on a network, such that all components cooperate together to perform a single or small set of related tasks.

Why build a distributed system? There are lots of advantages including the ability to connect remote users with remote resources in an open and scalable way. When we say *open*, we mean each component is continually open to interaction with other components. When we say *scalable*, we mean the system can easily be altered to accommodate changes in the number of users, resources and computing entities.

Thus, a distributed system can be much larger and more powerful given the combined capabilities of the distributed components, than combinations of stand-alone systems. But it's not easy - for a distributed system to be useful, it must be reliable. This is a difficult goal to

achieve because of the complexity of the interactions between simultaneously running components.

To be truly reliable, a distributed system must have the following characteristics:

- **Fault-Tolerant:** It can recover from component failures without performing incorrect actions.
- **Highly Available:** It can restore operations, permitting it to resume providing services even when some components have failed.
- **Recoverable:** Failed components can restart themselves and rejoin the system, after the cause of failure has been repaired.
- **Consistent:** The system can coordinate actions by multiple components often in the presence of concurrency and failure. This underlies the ability of a distributed system to act like a non-distributed system.
- **Scalable:** It can operate correctly even as some aspect of the system is scaled to a larger size. For example, we might increase the size of the network on which the system is running. This increases the frequency of network outages and could degrade a "non-scalable" system. Similarly, we might increase the number of users or servers, or overall load on the system. In a scalable system, this should not have a significant effect.
- **Predictable Performance:** The ability to provide desired responsiveness in a timely manner.
- **Secure:** The system authenticates access to data and services [1]

These are high standards, which are challenging to achieve. Probably the most difficult challenge is a distributed system must be able to continue operating correctly even when components fail. This issue is discussed in the following excerpt of an interview with Ken Arnold. Ken is a research scientist at Sun and is one of the original architects of Jini, and was a member of the architectural team that designed CORBA.

Failure is the defining difference between distributed and local programming, so you have to design distributed systems with the expectation of failure. Imagine asking people, "If the probability of something happening is one in 10^{13} , how often would it happen?" Common sense would be to answer, "Never." That is an infinitely large number in human terms. But if you ask a physicist, she would say, "All the time. In a cubic foot of air, those things happen all the time."

When you design distributed systems, you have to say, "Failure happens all the time." So when you design, you design for failure. It is your number one concern. What does designing for failure mean? One classic problem is partial failure. If I send a message to you and then a network failure occurs, there are two possible outcomes. One is that the message got to you, and then the network broke, and I just didn't get the response. The other is the message never got to you because the network broke before it arrived.

So if I never receive a response, how do I know which of those two results happened? I cannot determine that without eventually finding you. The network has to be repaired or you have to come up, because maybe what happened was not a network failure but you died. How does this change how I design things? For one thing, it puts a multiplier on the value of simplicity. The more things I can do with you, the more things I have to think about recovering from. [2]

Handling failures is an important theme in distributed systems design. Failures fall into two obvious categories: hardware and software. Hardware failures were a dominant concern until the late 80's, but since then internal hardware reliability has improved enormously. Decreased heat production and power consumption of smaller circuits, reduction of off-chip connections and wiring, and high-quality manufacturing techniques have all played a positive role in improving hardware reliability. Today, problems are most often associated with connections and mechanical devices, i.e., network failures and drive failures.

Software failures are a significant issue in distributed systems. Even with rigorous testing, software bugs account for a substantial fraction of unplanned downtime (estimated at 25-35%). Residual bugs in mature systems can be classified into two main categories [5].

- Heisenbug: A bug that seems to disappear or alter its characteristics when it is observed or researched. A common example is a bug that occurs in a release-mode compile of a program, but not when researched under debug-mode. The name "heisenbug" is a pun on the "Heisenberg uncertainty principle," a quantum physics term which is commonly (yet inaccurately) used to refer to the way in which observers affect the measurements of the things that they are observing, by the act of observing alone (this is actually the observer effect, and is commonly confused with the Heisenberg uncertainty principle).
- Bohrbug: A bug (named after the Bohr atom model) that, in contrast to a heisenbug, does not disappear or alter its characteristics when it is researched. A Bohrbug typically manifests itself reliably under a well-defined set of conditions. [6]

Heisenbugs tend to be more prevalent in distributed systems than in local systems. One reason for this is the difficulty programmers have in obtaining a coherent and comprehensive view of the interactions of concurrent processes.

Let's get a little more specific about the types of failures that can occur in a distributed system:

- Halting failures: A component simply stops. There is no way to detect the failure except by timeout: it either stops sending "I'm alive" (heartbeat) messages or fails to respond to requests. Your computer freezing is a halting failure.
- Fail-stop: A halting failure with some kind of notification to other components. A network file server telling its clients it is about to go down is a fail-stop.
- Omission failures: Failure to send/receive messages primarily due to lack of buffering space, which causes a message to be discarded with no notification to either the sender or receiver. This can happen when routers become overloaded.

- Network failures: A network link breaks.
- Network partition failure: A network fragments into two or more disjoint sub-networks within which messages can be sent, but between which messages are lost. This can occur due to a network failure.
- Timing failures: A temporal property of the system is violated. For example, clocks on different computers which are used to coordinate processes are not synchronized; when a message is delayed longer than a threshold period, etc.
- Byzantine failures: This captures several types of faulty behaviors including data corruption or loss, failures caused by malicious programs, etc. [1]

Our goal is to design a distributed system with the characteristics listed above (fault-tolerant, highly available, recoverable, etc.), which means we must design for failure. To design for failure, we must be careful to not make any assumptions about the reliability of the components of a system.

Everyone, when they first build a distributed system, makes the following eight assumptions. These are so well-known in this field that they are commonly referred to as the "8 Fallacies".

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous. [3]

Latency: the time between initiating a request for data and the beginning of the actual data transfer.

Bandwidth: A measure of the capacity of a communications channel. The higher a channel's bandwidth, the more information it can carry.

Topology: The different configurations that can be adopted in building networks, such as a ring, bus, star or meshed.

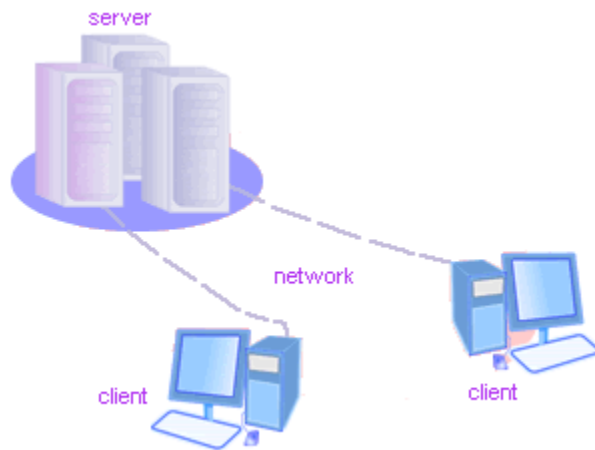
Homogeneous network: A network running a single network protocol.

So How Is It Done?

Building a reliable system that runs over an unreliable communications network seems like an impossible goal. We are forced to deal with uncertainty. A process knows its own state, and it knows what state other processes were in recently. But the processes have no way of knowing each other's current state. They lack the equivalent of shared memory. They also lack accurate ways to detect failure, or to distinguish a local software/hardware failure from a communication failure.

Distributed systems design is obviously a challenging endeavor. How do we do it when we are not allowed to assume anything, and there are so many complexities? We start by limiting the

scope. We will focus on a particular type of distributed systems design, one that uses a client-server model with mostly standard protocols. It turns out that these standard protocols provide considerable help with the low-level details of reliable network communications, which makes our job easier. Let's start by reviewing client-server technology and the protocols.



In client-server applications, the *server* provides some service, such as processing database queries or sending out current stock prices. The *client* uses the service provided by the server, either displaying database query results to the user or making stock purchase recommendations to an investor. The communication that occurs between the client and the server must be reliable. That is, no data can be dropped and it must arrive on the client side in the same order in which the server sent it.

There are many types of servers we encounter in a distributed system. For example, *file servers* manage disk storage units on which file systems reside. *Database servers* house databases and make them available to clients. *Network name servers* implement a mapping between a symbolic name or a service description and a value such as an IP address and port number for a process that provides the service.

In distributed systems, there can be many servers of a particular type, e.g., multiple file servers or multiple network name servers. The term *service* is used to denote a set of servers of a particular type. We say that a *binding* occurs when a process that needs to access a service becomes associated with a particular server which provides the service. There are many binding policies that define how a particular server is chosen. For example, the policy could be based on locality (a Unix NIS client starts by looking first for a server on its own machine); or it could be based on load balance (a CICS client is bound in such a way that uniform responsiveness for all clients is attempted).

A distributed service may employ *data replication*, where a service maintains multiple copies of data to permit local access at multiple locations, or to increase availability when a server process may have crashed. *Caching* is a related concept and very common in distributed systems. We say a process has cached data if it maintains a copy of the data locally, for quick access if it is needed again. A *cache hit* is when a request is satisfied from cached data, rather

than from the primary service. For example, browsers use document caching to speed up access to frequently used documents.

Caching is similar to replication, but cached data can become stale. Thus, there may need to be a policy for validating a cached data item before using it. If a cache is actively refreshed by the primary service, caching is identical to replication. [1]

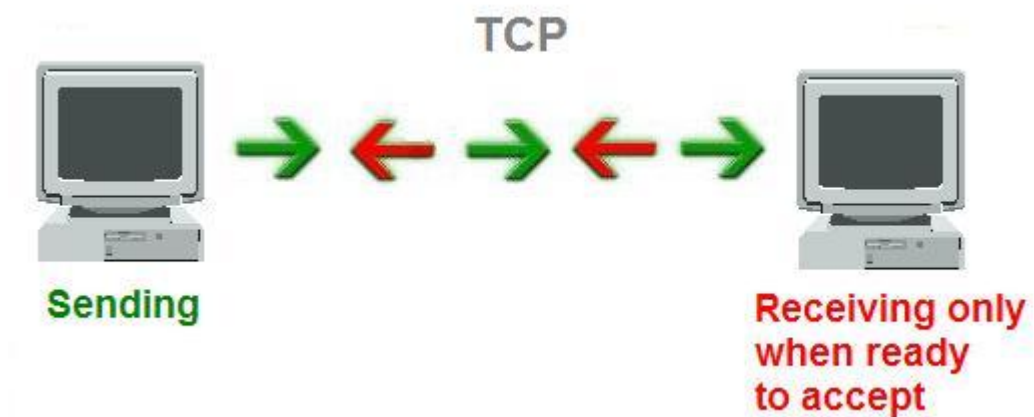
As mentioned earlier, the communication between client and server needs to be reliable. You have probably heard of *TCP/IP* before. The Internet Protocol (IP) suite is the set of communication protocols that allow for communication on the Internet and most commercial networks. The Transmission Control Protocol (TCP) is one of the core protocols of this suite. Using TCP, clients and servers can create connections to one another, over which they can exchange data in packets. The protocol guarantees reliable and in-order delivery of data from sender to receiver.

The IP suite can be viewed as a set of layers, each layer having the property that it only uses the functions of the layer below, and only exports functionality to the layer above. A system that implements protocol behavior consisting of layers is known as a *protocol stack*. Protocol stacks can be implemented either in hardware or software, or a mixture of both. Typically, only the lower layers are implemented in hardware, with the higher layers being implemented in software.

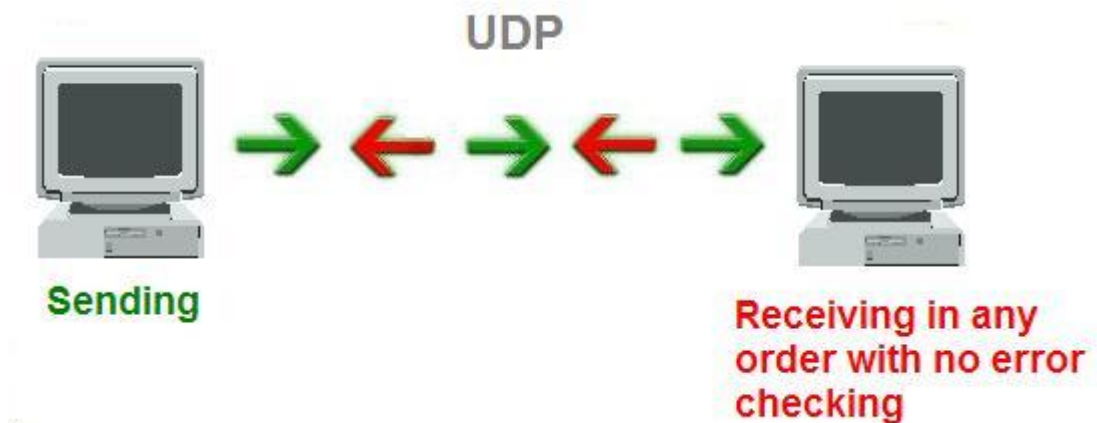
Resource : The history of TCP/IP mirrors the evolution of the Internet. [Here](#) is a brief overview of this history.

There are four layers in the IP suite:

1. *Application Layer* : The application layer is used by most programs that require network communication. Data is passed down from the program in an application-specific format to the next layer, then encapsulated into a transport layer protocol. Examples of applications are HTTP, FTP or Telnet.
2. *Transport Layer* : The transport layer's responsibilities include end-to-end message transfer independent of the underlying network, along with error control, fragmentation and flow control. End-to-end message transmission at the transport layer can be categorized as either *connection-oriented* (TCP) or *connectionless* (UDP). TCP is the more sophisticated of the two protocols, providing reliable delivery. First, TCP ensures that the receiving computer is ready to accept data. It uses a three-packet handshake in which both the sender and receiver agree that they are ready to communicate. Second, TCP makes sure that data gets to its destination. If the receiver doesn't acknowledge a particular packet, TCP automatically retransmits the packet typically three times. If necessary, TCP can also split large packets into smaller ones so that data can travel reliably between source and destination. TCP drops duplicate packets and rearranges packets that arrive out of sequence.



<>UDP is similar to TCP in that it is a protocol for sending and receiving packets across a network, but with two major differences. First, it is connectionless. This means that one program can send off a load of packets to another, but that's the end of their relationship. The second might send some back to the first and the first might send some more, but there's never a solid connection. UDP is also different from TCP in that it doesn't provide any sort of guarantee that the receiver will receive the packets that are sent in the right order. All that is guaranteed is the packet's contents. This means it's a lot faster, because there's no extra overhead for error-checking above the packet level. For this reason, games often use this protocol. In a game, if one packet for updating a screen position goes missing, the player will just jerk a little. The other packets will simply update the position, and the missing packet - although making the movement a little rougher - won't change anything.



<>Although TCP is more reliable than UDP, the protocol is still at risk of failing in many ways. TCP uses acknowledgements and retransmission to detect and repair loss. But it cannot overcome longer communication outages that disconnect the sender and receiver for long enough to defeat the retransmission strategy. The normal maximum disconnection time is between 30 and 90 seconds. TCP could signal a failure and give up when both end-points are fine. This is just one example of how TCP can fail, even though it does provide some mitigating strategies.

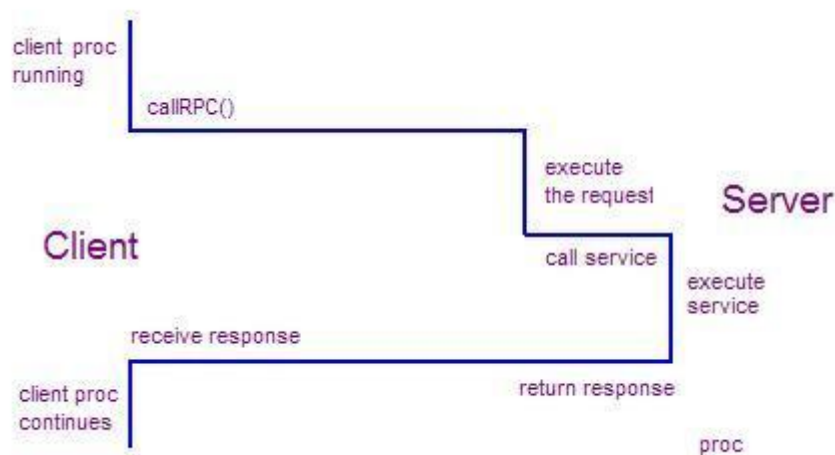
3. *Network Layer* : As originally defined, the Network layer solves the problem of getting packets across a single network. With the advent of the concept of internetworking, additional functionality was added to this layer, namely getting data from a source network to a destination network. This generally involves routing the packet across a network of networks, e.g. the Internet. IP performs the basic task of getting packets of data from source to destination.
4. *Link Layer* : The link layer deals with the physical transmission of data, and usually involves placing frame headers and trailers on packets for travelling over the physical network and dealing with physical components along the way.

Resource : For more information on the IP Suite, refer to the [Wikipedia article](#).

Remote Procedure Calls

Many distributed systems were built using TCP/IP as the foundation for the communication between components. Over time, an efficient method for clients to interact with servers evolved called RPC, which means *remote procedure call*. It is a powerful technique based on extending the notion of local procedure calling, so that the called procedure may not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them.

An RPC is similar to a function call. Like a function call, when an RPC is made, the arguments are passed to the remote procedure and the caller waits for a response to be returned. In the illustration below, the client makes a procedure call that sends a request to the server. The client process waits until either a reply is received, or it times out. When the request arrives at the server, it calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client process continues.



<>

Threads are common in RPC-based distributed systems. Each incoming request to a server typically spawns a new thread. A thread in the client typically issues an RPC and then blocks (waits). When the reply is received, the client thread resumes execution.

A programmer writing RPC-based code does three things:

1. Specifies the protocol for client-server communication
2. Develops the client program
3. Develops the server program

The communication protocol is created by *stubs* generated by a protocol compiler. A stub is a routine that doesn't actually do much other than declare itself and the parameters it accepts. The stub contains just enough code to allow it to be compiled and linked.

The client and server programs must communicate via the procedures and data types specified in the protocol. The server side registers the procedures that may be called by the client and receives and returns data required for processing. The client side calls the remote procedure, passes any required data and receives the returned data.

Thus, an RPC application uses classes generated by the stub generator to execute an RPC and wait for it to finish. The programmer needs to supply classes on the server side that provide the logic for handling an RPC request.

RPC introduces a set of error cases that are not present in local procedure programming. For example, a *binding error* can occur when a server is not running when the client is started. *Version mismatches* occur if a client was compiled against one version of a server, but the server has now been updated to a newer version. A timeout can result from a server crash, network problem, or a problem on a client computer.

Some RPC applications view these types of errors as unrecoverable. Fault-tolerant systems, however, have alternate sources for critical services and *fail-over* from a primary server to a backup server.

A challenging error-handling case occurs when a client needs to know the outcome of a request in order to take the next step, after failure of a server. This can sometimes result in incorrect actions and results. For example, suppose a client process requests a ticket-selling server to check for a seat in the orchestra section of Carnegie Hall. If it's available, the server records the request and the sale. But the request fails by timing out. Was the seat available and the sale recorded? Even if there is a backup server to which the request can be re-issued, there is a risk that the client will be sold two tickets, which is an expensive mistake in Carnegie Hall [1].

Here are some common error conditions that need to be handled:

- **Network data loss resulting in retransmit:** Often, a system tries to achieve 'at most once' transmission tries. In the worst case, if duplicate transmissions occur, we try to minimize any damage done by the data being received multiple time.
- **Server process crashes during RPC operation:** If a server process crashes before it completes its task, the system usually recovers correctly because the client will initiate a retry request once the server has recovered. If the server crashes completing the task but before the RPC reply is sent, duplicate requests sometimes result due to client retries.
- **Client process crashes before receiving response:** Client is restarted. Server discards response data.

Some Distributed Design Principles

Given what we have covered so far, we can define some fundamental design principles which every distributed system designer and software engineer should know. Some of these may seem obvious, but it will be helpful as we proceed to have a good starting list.

- As Ken Arnold says: "You have to design distributed systems with the expectation of failure." Avoid making assumptions that any component in the system is in a particular state. A classic error scenario is for a process to send data to a process running on a second machine. The process on the first machine receives some data back and processes it, and then sends the results back to the second machine assuming it is ready to receive. Any number of things could have failed in the interim and the sending process must anticipate these possible failures.
- Explicitly define failure scenarios and identify how likely each one might occur. Make sure your code is thoroughly covered for the most likely ones.
- Both clients and servers must be able to deal with unresponsive senders/receivers.
- Think carefully about how much data you send over the network. Minimize traffic as much as possible.
- Latency is the time between initiating a request for data and the beginning of the actual data transfer. Minimizing latency sometimes comes down to a question of whether you should make many little calls/data transfers or one big call/data transfer. The way to make this decision is to experiment. Do small tests to identify the best compromise.
- Don't assume that data sent across a network (or even sent from disk to disk in a rack) is the same data when it arrives. If you must be sure, do checksums or validity checks on data to verify that the data has not changed.
- Caches and replication strategies are methods for dealing with state across components. We try to minimize stateful components in distributed systems, but it's challenging. State is something held in one place on behalf of a process that is in another place, something that cannot be reconstructed by any other component. If it can be reconstructed it's a cache. Caches can be helpful in mitigating the risks of maintaining state across components. But cached data can become stale, so there may need to be a policy for validating a cached data item before using it.

If a process stores information that can't be reconstructed, then problems arise. One possible question is, "Are you now a single point of failure?" I have to talk to *you* now - I can't talk to anyone else. So what happens if you go down? To deal with this issue, you could be replicated. Replication strategies are also useful in mitigating the risks of maintaining state. But there are challenges here too: What if I talk to one replicant and modify some data, then I talk to another? Is that modification guaranteed to have

already arrived at the other? What happens if the network gets partitioned and the replicants can't talk to each other? Can anybody proceed?

There are a set of tradeoffs in deciding how and where to maintain state, and when to use caches and replication. It's more difficult to run small tests in these scenarios because of the overhead in setting up the different mechanisms.

- Be sensitive to speed and performance. Take time to determine which parts of your system can have a significant impact on performance: Where are the bottlenecks and why? Devise small tests you can do to evaluate alternatives. Profile and measure to learn more. Talk to your colleagues about these alternatives and your results, and decide on the best solution.
- Acks are expensive and tend to be avoided in distributed systems wherever possible.
- Retransmission is costly. It's important to experiment so you can tune the delay that prompts a retransmission to be optimal.

Exercises

1. Have you ever encountered a Heisenbug? How did you isolate and fix it?
2. For the different failure types listed above, consider what makes each one difficult for a programmer trying to guard against it. What kinds of processing can be added to a program to deal with these failures?
3. Explain why each of the 8 fallacies is actually a fallacy.
4. Contrast TCP and UDP. Under what circumstances would you choose one over the other?
5. What's the difference between caching and data replication?
6. What are stubs in an RPC implementation?
7. What are some of the error conditions we need to guard against in a distributed environment that we do not need to worry about in a local programming environment?
8. Why are pointers (references) not usually passed as parameters to a Remote Procedure Call?
9. Here is an interesting problem called *partial connectivity* that can occur in a distributed environment. Let's say A and B are systems that need to talk to each other. C is a master that also talks to A and B individually. The communications between A and B fail. C can tell that A and B are both healthy. C tells A to send something to B and waits for this to occur. C has no way of knowing that A cannot talk to B, and thus waits and waits and waits. What diagnostics can you add in your code to deal with this situation?
10. What is the *leader-election* algorithm? How can it be used in a distributed system?
11. This is the Byzantine Generals problem: Two generals are on hills either side of a valley. They each have an army of 1000 soldiers. In the woods in the valley is an enemy army of 1500 men. If each general attacks alone, his army will lose. If they

attack together, they will win. They wish to send messengers through the valley to coordinate when to attack. However, the messengers may get lost or caught in the woods (or brainwashed into delivering different messages). How can they devise a scheme by which they either attack with high probability, or not at all?

References

- [1] Birman, Kenneth. **Reliable Distributed Systems: Technologies, Web Services and Applications**. New York: Springer-Verlag, 2005.
- [2] [Interview with Ken Arnold](#)
- [3] [The Eight Fallacies](#)
- [4] [Wikipedia article on IP Suite](#)
- [5] Gray, J. and Reuter, A. **Transaction Processing: Concepts and Techniques**. San Mateo, CA: Morgan Kaufmann, 1993.
- [6] [Bohrbugs and Heisenbugs](#)

I Flickr Architecture



Wed, 08/29/2007 - 10:04 — [Todd Hoff](#)

- [Flickr Architecture \(1164\)](#)

Flickr is both my favorite [bird](#) and the web's leading photo sharing site. Flickr has an amazing challenge, they must handle a vast sea of ever expanding new content, ever increasing legions of users, and a constant stream of new features, all while providing excellent performance. How do they do it?

Site: <http://www.flickr.com/>

Information Sources

- [Flickr and PHP](#) (an early document)
- [Capacity Planning](#) for LAMP
- [Federation at Flickr: A tour of the Flickr Architecture](#).

- [Building Scalable Web Sites](#) by Cal Henderson from Flickr.
- [Database War Stories #3](#): Flickr by Tim O'Reilly
- [Cal Henderson's Talks](#). A lot of useful PowerPoint presentations.

Platform

- PHP
- [MySQL](#)
- Shards
- [Memcached](#) for a caching layer.
- [Squid](#) in reverse-proxy for html and images.
- [Linux](#) (RedHat)
- Smarty for templating
- [Perl](#)
- PEAR for XML and Email parsing
- ImageMagick, for image processing
- [Java](#), for the node service
- [Apache](#)
- [SystemImager](#) for [deployment](#)
- [Ganglia](#) for distributed system monitoring
- [Subcon](#) stores essential system configuration files in a subversion repository for easy deployment to machines in a cluster.
- Cvsup for distributing and updating collections of files across a network.

The Stats

- More than 4 billion queries per day.
- ~35M photos in squid cache (total)

- ~2M photos in squid's RAM
- ~470M photos, 4 or 5 sizes of each
- 38k req/sec to memcached (12M objects)
- 2 PB raw storage (consumed about ~1.5TB on Sunday)
- Over 400,000 photos being added every day

The Architecture

- A pretty picture of Flickr's architecture can be found on this [slide](#) . A simple depiction is:

-- Pair of ServerIron's

---- Squid Caches

----- Net App's

---- PHP App Servers

----- Storage Manager

----- **Master**-master shards

----- Dual Tree Central Database

----- Memcached Cluster

----- Big Search Engine

- The Dual Tree structure is a custom set of changes to MySQL that allows scaling by incrementally adding masters without a ring architecture. This allows cheaper scaling because you need less hardware as compared to master-master setups which always requires double the hardware.

- The central database includes data like the 'users' table, which includes primary user keys (a few different IDs) and a pointer to which shard a users' data can be found on.

- Use dedicated servers for static content.

- Talks about how to support Unicode.
- Use a share nothing architecture.
- Everything (except photos) are stored in the database.
- Statelessness means they can bounce people around servers and it's easier to make their APIs.
- Scaled at first by replication, but that only helps with reads.
- Create a search farm by replicating the portion of the database they want to search.
- Use horizontal scaling so they just need to add more machines.
- Handle pictures emailed from users by parsing each **email** as it's delivered in PHP. Email is parsed for any photos.
- Earlier they suffered from Master-**Slave** lag. Too much load and they had a single point of failure.
- They needed the ability to make live maintenance, repair data, and so forth, without taking the site down.
- Lots of excellent material on **capacity** planning. Take a look in the Information Sources for more details.
- Went to a federated approach so they can scale far into the future:
 - Shards: My data gets stored on my shard, but the record of performing action on your comment, is on your shard. When making a comment on someone else's' blog
 - Global Ring: Its like DNS, you need to know where to go and who controls where you go. Every page view, calculate where your data is, at that moment of time.
 - PHP logic to connect to the shards and keep the data consistent (10 lines of code with comments!)
- Shards:
 - Slice of the main database
 - Active Master-Master Ring Replication: a few drawbacks in MySQL 4.1, as honoring

commits in Master-Master. AutoIncrement IDs are automated to keep it Active Active.

- **Shard** assignments are from a random number for new accounts

- Migration is done from time to time, so you can remove certain power users. Needs to be balanced if you have a lot of photos... 192,000 photos, 700,000 tags, will take about 3-4 minutes. Migration is done manually.

- Clicking a Favorite:

- Pulls the Photo owners Account from Cache, to get the shard location (say on shard-5)

- Pulls my Information from cache, to get my shard location (say on shard-13)

- Starts a "distributed transaction" - to answer the question: Who favorited the photo?

What are my favorites?

- Can ask question from any shard, and recover data. Its absolutely redundant.

- To get rid of replication lag...

- every page load, the user is assigned to a bucket

- if host is down, go to next host in the list; if all hosts are down, display an error page.

They don't use persistent connections, they build connections and tear it down. Every page load thus, tests the connection.

- Every users reads and writes are kept in one shard. Notion of replication lag is gone.

- Each server in shard is 50% loaded. Shut down 1/2 the servers in each shard. So 1 server in the shard can take the full load if a server of that shard is down or in maintenance mode. To upgrade you just have to shut down half the shard, upgrade that half, and then repeat the process.

- Periods of time when traffic spikes, they break the 50% rule though. They do something like 6,000-7,000 queries per second. Now, its designed for at most 4,000 queries per second to keep it at 50% load.

- Average queries per page, are 27-35 SQL statements. Favorites counts are real time.

API access to the database is all real time. Achieved the real time requirements without

any disadvantages.

- Over 36,000 queries per second - running within capacity threshold. Burst of traffic, double 36K/qps.

- Each Shard holds 400K+ users data.

- A lot of data is stored twice. For example, a comment is part of the relation between the commentor and the commentee. Where is the comment stored? How about both places?

Transactions are used to prevent out of sync data: open transaction 1, write commands, open transaction 2, write commands, commit 1st transaction if all is well, commit 2nd transaction if 1st committed. but there still a chance for failure when a box goes down during the 1st commit.

- Search:

- Two search back-ends: shards 35k qps on a few shards and Yahoo!'s (proprietary) web search

- Owner's single tag search or a batch tag change (say, via Organizr) goes to the Shards due to real-time requirements, everything else goes to Yahoo!'s engine (probably about 90% behind the real-time goodness)

- Think of it such that you've got **Lucene**-like search

- Hardware:

- EMT64 w/RHEL4, 16GB RAM

- 6-disk 15K RPM **RAID**-10.

- Data size is at 12 TB of user metadata (these are not photos, this is just innodb ibdata files - the photos are a lot larger).

- 2U boxes. Each shard has ~120GB of data.

- Backup procedure:

- ibbackup on a cron job, that runs across various shards at different times. Hotbackup to a spare.

- Snapshots are taken every night across the entire cluster of databases.
- Writing or deleting several huge backup files at once to a replication filestore can wreck performance on that filestore for the next few hours as it replicates the backup files. Doing this to an in-production photo storage filer is a bad idea.
- However much it costs to keep multiple days of backups of all of your data, it's worth it. Keeping staggered backups is good for when you discover something gone wrong a few days later. something like 1, 2, 10 and 30 day backups.
- Photos are stored on the filer. Upon upload, it processes the photos, gives you different sizes, then its complete. Metadata and points to the filers, are stored in the database.
- Aggregating the data: Very fast, because its a process per shard. Stick it into a table, or recover data from another copy from other users shards.
- max_connections = 400 connections per shard, or 800 connections per server & shard. Plenty of capacity and connections. Thread cache is set to 45, because you don't have more than 45 users having simultaneous activity.
- Tags:
 - Tags do not fit well with traditional normalized RDBMs schema design. Denormalization or heavy caching is the only way to generate a tag cloud in milliseconds for hundreds of millions of tags.
 - Some of their data views are calculated offline by dedicated processing clusters which save the results into MySQL because some relationships are so complicated to calculate it would absorb all the database CPU cycles.
- Future Direction:
 - Make it faster with real-time **BCP**, so all data centers can receive writes to the data layer (db, memcache, etc) all at the same time. Everything is active nothing will ever be idle.

Lessons Learned

- **Think of your application as more than just a web application.** You'll have REST APIs, SOAP APIs, RSS feeds, Atom feeds, etc.
- **Go stateless.** Statelessness makes for a simpler more robust system that can handle upgrades without flinching.
- **Re-architecting your database sucks.**
- **Capacity plan.** Bring capacity planning into the product discussion EARLY. Get buy-in from the \$\$\$ people (and engineering management) that it's something to watch.
- **Start slow.** Don't buy too much equipment just because you're scared/happy that your site will explode.
- **Measure reality.** Capacity planning math should be based on real things, not abstract ones.
- **Build in logging and metrics.** Usage stats are just as important as server stats. Build in custom metrics to measure real-world usage to server-based stats.
- **Cache.** **Caching** and RAM is the answer to everything.
- **Abstract.** Create clear levels of abstraction between database work, business logic, page logic, page mark-up and the presentation layer. This supports quick turn around iterative development.
- **Layer.** Layering allows developers to create page level logic which designers can use to build the user experience. Designers can ask for page logic as needed. It's a negotiation between the two parties.
- **Release frequently.** Even every 30 minutes.
- **Forget about small efficiencies,** about 97% of the time. Premature optimization is the root of all evil.
- **Test in production.** Build into the architecture mechanisms (config flags, load

balancing, etc.) with which you can deploy new hardware easily into (and out of) production.

- **Forget benchmarks.** Benchmarks are fine for getting a general idea of capabilities, but not for planning. Artificial tests give artificial results, and the time is better used with testing for real.

- **Find ceilings.**

- What is the maximum something that every server can do ?

- How close are you to that maximum, and how is it trending ?

- MySQL (disk IO ?)

- SQUID (disk IO ? or CPU ?)

- memcached (CPU ? or network ?)

- **Be sensitive to the usage patterns for your type of application.**

- Do you have event related growth? For example: disaster, news event.

- Flickr gets 20-40% more uploads on first work day of the year than any previous peak the previous year.

- 40-50% more uploads on Sundays than the rest of the week, on average

- **Be sensitive to the demands of exponential growth.** More users means more content, more content means more connections, more connections mean more usage.

- **Plan for peaks.** Be able to handle peak loads up and down the stack.

- [Apache](#)

- [Example](#)

- [Java](#)

- [Linux](#)

- [MySQL](#)

- [Perl](#)

- [PHP](#)

- [Shard](#)
- [Visit Flickr Architecture](#)
- 24401 reads

Comments

Wed, 08/08/2007 - 13:23 — Sam (not verified)

How to store images?

Is there an easier solution managing images in combination of database and files? It seems storing your images in database might really slow down the site.

- [reply](#)

Wed, 08/08/2007 - 16:23 — [Douglas F Shearer](#) (not verified)

RE: How to store images?

Flickr only store a reference to an image in their databases, the actual file is stored on a separate storage server elsewhere on the network.

A typical URL for a Flickr image looks like this:

http://farm1.static.flickr.com/104/301293250_dc284905d0_m.jpg

If we split this up we get:

farm1 - Obviously the farm at which the image is stored. I have yet to see a value other than one.

.static.flickr.com - Fairly self explanatory.

/104 - The server ID number.

/301293250 - The image ID.

_dc284905d0 - The image 'secret'. I assume this is to prevent images being copied without first getting the information from the API.

_m - The size of the image. In this case the 'm' denotes medium, but this can be small, thumb etc. For the standard image size there is no size of this form in the URL.

I Amazon Architecture



Tue, 09/18/2007 - 19:44 — Todd Hoff

- [Amazon Architecture \(2495\)](#)

This is a wonderfully informative Amazon update based on Joachim Rohde's discovery of an interview with Amazon's CTO. You'll learn about how Amazon organizes their teams around services, the CAP theorem of building scalable systems, how they deploy software, and a lot more. Many new additions from the ACM Queue article have also been included.

Amazon grew from a tiny online bookstore to one of the largest stores on earth. They did it while pioneering new and interesting ways to rate, review, and recommend products.

Greg Linden shared his version of Amazon's birth pangs in a series of blog articles

Site: <http://amazon.com>

Information Sources

- [Early Amazon by Greg Linden](#)
- [How Linux](#) saved Amazon millions
- [Interview Werner Vogels](#) - Amazon's CTO

- [Asynchronous Architectures](#) - a nice [summary](#) of Werner Vogels' talk by Chris Loosley
- [Learning from the Amazon technology platform](#) - A Conversation with Werner Vogels
- [Werner Vogels' Weblog](#) - building scalable and robust distributed systems

Platform

- Linux
- Oracle
- C++
- Perl
- Mason
- Java
- Jboss
- Servlets

The Stats

- More than 55 million active customer accounts.
- More than 1 million active retail partners worldwide.
- Between 100-150 services are accessed to build a page.

The Architecture

- What is it that we really mean by scalability? A service is said to be scalable if when we increase the resources in a system, it results in increased performance in a manner proportional to resources added. Increasing performance in general means serving more units of work, but it can also be to handle larger units of work, such as when datasets grow.
- The big architectural change that Amazon made was to move from a two-tier

monolith to a fully-distributed, decentralized, services platform serving many different applications.

- Started as one application talking to a back end. Written in C++.
- It grew. For years the scaling efforts at Amazon focused on making the back-end databases scale to hold more items, more customers, more orders, and to support multiple international sites. In 2001 it became clear that the front-end application couldn't scale anymore. The databases were split into small parts and around each part and created a services interface that was the only way to access the data.
- The databases became a shared resource that made it hard to scale-out the overall business. The front-end and back-end processes were restricted in their evolution because they were shared by many different teams and processes.
- Their architecture is loosely coupled and built around services. A service-oriented architecture gave them the isolation that would allow building many software components rapidly and independently.
- Grew into hundreds of services and a number of application servers that aggregate the information from the services. The application that renders the Amazon.com Web pages is one such application server. So are the applications that serve the Web-services interface, the customer service application, and the seller interface.
- Many third party technologies are hard to scale to Amazon size. Especially communication infrastructure technologies. They work well up to a certain scale and then fail. So they are forced to build their own.
- Not stuck with one particular approach. Some places they use jboss/java, but they use only servlets, not the rest of the J2EE stack.
- C++ is used to process requests. Perl/Mason is used to build content.
- Amazon doesn't like middleware because it tends to be framework and not a tool. If you use a middleware package you get lock-in around the software patterns they have

chosen. You'll only be able to use their software. So if you want to use different packages you won't be able to. You're stuck. One event loop for messaging, data persistence, **AJAX**, etc. Too complex. If middleware was available in smaller components, more as a tool than a framework, they would be more interested.

- The SOAP web stack seems to want to solve all the same distributed systems problems all over again.
- Offer both SOAP and REST web services. 30% use SOAP. These tend to be Java and .NET users and use WSDL files to generate remote object interfaces. 70% use REST. These tend to be **PHP** or PERL users.
- In either SOAP or REST developers can get an object interface to Amazon. Developers just want to get job done. They don't care what goes over the wire.
- Amazon wanted to build an open community around their services. Web services were chosen because it's simple. But that's only on the perimeter. Internally it's a service oriented architecture. You can only access the data via the interface. It's described in WSDL, but they use their own encapsulation and transport mechanisms.
- Teams are Small and are Organized Around Services
 - Services are the independent units delivering functionality within Amazon. It's also how Amazon is organized internally in terms of teams.
 - If you have a new business idea or problem you want to solve you form a team. Limit the team to 8-10 people because communication hard. They are called two pizza teams. The number of people you can feed off two pizzas.
 - Teams are small. They are assigned authority and empowered to solve a problem as a service in anyway they see fit.
 - As an example, they created a team to find phrases within a book that are unique to the text. This team built a separate service interface for that feature and they had authority to do what they needed.

- Extensive A/B testing is used to integrate a new service . They see what the impact is and take extensive measurements.

- Deployment

- They create special infrastructure for managing dependencies and doing a **deployment**.

- Goal is to have all right services to be deployed on a box. All application code, monitoring, licensing, etc should be on a box.

- Everyone has a home grown system to solve these problems.

- Output of deployment process is a virtual machine. You can use **EC2** to run them.

- Work From the Customer Backwards to Verify a New Service is Worth Doing

- Work from the customer backward. Focus on value you want to deliver

for the customer.

- Force developers to focus on value delivered to the customer instead of building technology first and then figuring how to use it.

- Start with a press release of what features the user will see and work backwards to check that you are building something valuable.

- End up with a design that is as minimal as possible. Simplicity is the key if you really want to build large distributed systems.

- State Management is the Core Problem for Large Scale Systems

- Internally they can deliver infinite storage.

- Not all that many operations are stateful. Checkout steps are stateful.

- Most recent clicked web page service has recommendations based on session IDs.

- They keep track of everything anyway so it's not a matter of keeping state. There's little separate state that needs to be kept for a session. The services will already be keeping the information so you just use the services.

- Eric Brewer's CAP Theorem or the Three properties of Systems

- Three properties of a system: consistency, availability, tolerance to network partitions.

- You can have at most two of these three properties for any shared-data system.
- Partitionability: divide nodes into small groups that can see other groups, but they can't see everyone.
- Consistency: write a value and then you read the value you get the same value back.
In a partitioned system there are windows where that's not true.
- Availability: may not always be able to write or read. The system will say you can't write because it wants to keep the system consistent.
- To scale you have to partition, so you are left with choosing either high consistency or high availability for a particular system. You must find the right overlap of availability and consistency.
- Choose a specific approach based on the needs of the service.
- For the checkout process you always want to honor requests to add items to a shopping cart because it's revenue producing. In this case you choose high availability. Errors are hidden from the customer and sorted out later.
- When a customer submits an order you favor consistency because several services--credit card processing, shipping and handling, reporting--are simultaneously accessing the data.

Lessons Learned

- You must change your mentality to build really scalable systems. Approach chaos in a probabilistic sense that things will work well. In traditional systems we present a perfect world where nothing goes down and then we build complex algorithms (agreement technologies) on this perfect world. Instead, take it for granted stuff fails, that's reality, embrace it. For example, go more with a fast reboot and fast recover approach. With a decent spread of data and services you might get close to 100%. Create

self-healing, self-organizing lights out operations.

- Create a shared nothing infrastructure. Infrastructure can become a shared resource for development and deployment with the same downsides as shared resources in your logic and data tiers. It can cause locking and blocking and dead lock. A service oriented architecture allows the creation of a parallel and isolated development process that scales feature development to match your growth.
- Open up your system with APIs and you'll create an ecosystem around your application.
- Only way to manage as large distributed system is to keep things as simple as possible. Keep things simple by making sure there are no hidden requirements and hidden dependencies in the design. Cut technology to the minimum you need to solve the problem you have. It doesn't help the company to create artificial and unneeded layers of complexity.
- Organizing around services gives agility. You can do things in parallel is because the output is a service. This allows fast time to market. Create an infrastructure that allows services to be built very fast.
- There's bound to be problems with anything that produces hype before real implementation
- Use SLAs internally to manage services.
- Anyone can very quickly add web services to their product. Just implement one part of your product as a service and start using it.
- Build your own infrastructure for performance, reliability, and cost control reasons. By building it yourself you never have to say you went down because it was company X's fault. Your software may not be more reliable than others, but you can fix, debug, and deployment much quicker than when working with a 3rd party.
- Use measurement and objective debate to separate the good from the bad. I've been

to several presentations by ex-Amazoners and this is the aspect of Amazon that strikes me as uniquely different and interesting from other companies. Their deep seated ethic is to expose real customers to a choice and see which one works best and to make decisions based on those tests.

Avinash

Kaushik calls this getting rid of the influence of the HiPPO's, the highest paid people in the room. This is done with techniques like A/B testing and Web Analytics. If you have a question about what you should do code it up, let people use it, and see which alternative gives you the results you want.

- Create a frugal culture. Amazon used doors for desks, for example.
- Know what you need. Amazon has a bad experience with an early recommender system that didn't work out: "This wasn't what Amazon needed. Book recommendations at Amazon needed to work from sparse data, just a few ratings or purchases. It needed to be fast. The system needed to scale to massive numbers of customers and a huge catalog. And it needed to enhance discovery, surfacing books from deep in the catalog that readers wouldn't find on their own."
- People's side projects, the one's they follow because they are interested, are often ones where you get the most value and innovation. Never underestimate the power of wandering where you are most interested.
- Involve everyone in making dog food. Go out into the warehouse and pack books during the Christmas rush. That's teamwork.
- Create a staging site where you can run thorough tests before releasing into the wild.
- A robust, clustered, replicated, distributed file system is perfect for read-only data used by the web servers.
- Have a way to rollback if an update doesn't work. Write the tools if necessary.
- Switch to a deep services-based architecture

(<http://webservices.sys-con.com/read/262024.htm>).

- Look for three things in interviews: enthusiasm, creativity, competence. The single biggest predictor of success at Amazon.com was enthusiasm.
- Hire a Bob. Someone who knows their stuff, has incredible debugging skills and system knowledge, and most importantly, has the stones to tackle the worst high pressure problems imaginable by just leaping in.
- Innovation can only come from the bottom. Those closest to the problem are in the best position to solve it. any organization that depends on innovation must embrace chaos. Loyalty and obedience are not your tools.
- Creativity must flow from everywhere.
- Everyone must be able to experiment, learn, and iterate. Position, obedience, and tradition should hold no power. For innovation to flourish, measurement must rule.
- Embrace innovation. In front of the whole company, Jeff Bezos would give an old Nike shoe as "Just do it" award to those who innovated.
- Don't pay for performance. Give good perks and high pay, but keep it flat. Recognize exceptional work in other ways. Merit pay sounds good but is almost impossible to do fairly in large organizations. Use non-monetary awards, like an old shoe. It's a way of saying thank you, somebody cared.
- Get big fast. The big guys like Barnes and Nobel are on your tail. Amazon wasn't even the first, second, or even third book store on the web, but their vision and drive won out in the end.
- In the data center, only 30 percent of the staff time spent on infrastructure issues related to value creation, with the remaining 70 percent devoted to dealing with the "heavy lifting" of hardware procurement, software management, load balancing, maintenance, scalability challenges and so on.
- Prohibit direct database access by clients. This means you can make you service scale

and be more reliable without involving your clients. This is much like Google's ability to independently distribute improvements in their stack to the benefit of all applications.

- Create a single unified service-access mechanism. This allows for the easy aggregation of services, decentralized request routing, distributed request tracking, and other advanced infrastructure techniques.

- Making Amazon.com available through a Web services interface to any developer in the world free of charge has also been a major success because it has driven so much innovation that they couldn't have thought of or built on their own.

- Developers themselves know best which tools make them most productive and which tools are right for the job.

- Don't impose too many constraints on engineers. Provide incentives for some things, such as integration with the monitoring system and other infrastructure tools. But for the rest, allow teams to function as independently as possible.

- Developers are like artists; they produce their best work if they have the freedom to do so, but they need good tools. Have many support tools that are of a self-help nature. Support an environment around the service development that never gets in the way of the development itself.

- You build it, you run it. This brings developers into contact with the day-to-day operation of their software. It also brings them into day-to-day contact with the customer. This customer feedback loop is essential for improving the quality of the service.

- Developers should spend some time with customer service every two years. Their they'll actually listen to customer service calls, answer customer service e-mails, and really understand the impact of the kinds of things they do as technologists.

- Use a "voice of the customer," which is a realistic story from a customer about some specific part of your site's experience. This helps managers and engineers connect with

the fact that we build these technologies for real people. Customer service statistics are an early indicator if you are doing something wrong, or what the real pain points are for your customers.

- Infrastructure for Amazon, like for Google, is a huge competitive advantage. They can build very complex applications out of primitive services that are by themselves relatively simple. They can scale their operation independently, maintain unparalleled system availability, and introduce new services quickly without the need for massive reconfiguration.

- [Example](#)
- [Java](#)
- [Linux](#)
- [Oracle](#)
- [Perl](#)
- [Visit Amazon Architecture](#)
- 33843 reads

Comments

Thu, 08/09/2007 - 16:22 — [herval](#) (not verified)

Jeff.. Bazos?

Jeff.. Bazos?

- [reply](#)

Wed, 08/29/2007 - 22:07 — Joachim Rohde (not verified)

Werner Vogels, the CTO of

Werner Vogels, the CTO of amazon, spoke a tiny bit about technical details on SE-Radio.

You can find the podcast under http://www.se-radio.net/index.php?post_id=157593

Interesting episode.

- [reply](#)

Fri, 08/31/2007 - 05:27 — [Todd Hoff](#)



Re: Amazon Architecture

That is a good interview. Thanks. I'll be adding the new information soon.

- [reply](#)

Fri, 09/07/2007 - 08:39 — [Arturo Fernandez](#) (not verified)

Re: Amazon Architecture

Amazon uses [Perl](#) and Mason.

See: <http://www.masonhq.com/?MasonPoweredSites>

- [reply](#)

Tue, 09/11/2007 - 07:52 — [Alexei A. Korolev](#) (not verified)

Re: Amazon Architecture

as i see they reduce c++ part and move to j2ee?

- [reply](#)

Tue, 09/18/2007 - 02:30 — Anonymous (not verified)

It's WSDL

I am not sure how you can get that one wrong, unless you are a manager, but even then some engineer would school you 'til Sunday.

- [reply](#)

Tue, 09/18/2007 - 04:37 — [Todd Hoff](#)



Re: It's WSDL

Thanks for catching that. I listen to these things a few times and sometimes I just write what I hear instead of what I mean.

- [reply](#)

Tue, 09/18/2007 - 07:28 — [Werner](#) (not verified)

Re: Amazon Architecture

I actually gave a scrisper definition of scalability at: [A Word on Scalability](#)

Personaly I like [the interview in ACM Queue](#) best for a high level view

--Werner

I Scaling Twitter: Making Twitter 10000 Percent Faster



Mon, 10/08/2007 - 21:01 — [Todd Hoff](#)

- [Scaling Twitter: Making Twitter 10000 Percent Faster \(913\)](#)

Twitter started as a side project and blew up fast, going from 0 to millions of page views within a few terrifying months. Early design decisions that worked well in the small melted under the crush of new users chirping tweets to all their friends. Web darling

Ruby on Rails was fingered early for the scaling problems, but Blaine Cook, Twitter's lead architect, held Ruby blameless:

For us, it's really about scaling horizontally - to that end, Rails and Ruby haven't been stumbling blocks, compared to any other language or framework. The performance boosts associated with a "faster" language would give us a 10-20% improvement, but thanks to architectural changes that Ruby and Rails happily accommodated, Twitter is 10000% faster than it was in January.

If Ruby on Rails wasn't to blame, how did Twitter learn to scale ever higher and higher?

Update: added slides Small Talk on Getting Big. Scaling a Rails App & all that Jazz

Site: <http://twitter.com>

Information Sources

- [Scaling Twitter Video](#) by Blaine Cook.
- [Scaling Twitter Slides](#)
- [Good News](#) blog post by Rick Denatale
- [Scaling Twitter](#) blog post Patrick Joyce.
- [Twitter API Traffic is 10x Twitter's Site.](#)
- [A Small Talk on Getting Big. Scaling a Rails App & all that Jazz](#) - really cute dog picks

The Platform

- Ruby on Rails
- Erlang
- **MySQL**
- **Mongrel** - hybrid Ruby/C HTTP server designed to be small, fast, and secure

- Munin
- Nagios
- Google Analytics
- AWStats - real-time logfile analyzer to get advanced statistics
- Memcached

The Stats

- Over 350,000 users. The actual numbers are as always, very super super top secret.
- 600 requests per second.
- Average 200-300 connections per second. Spiking to 800 connections per second.
- MySQL handled 2,400 requests per second.
- 180 Rails instances. Uses Mongrel as the "web" server.
- 1 MySQL Server (one big 8 core box) and 1 slave. Slave is read only for statistics and reporting.
- 30+ processes for handling odd jobs.
- 8 Sun X4100s.
- Process a request in 200 milliseconds in Rails.
- Average time spent in the database is 50-100 milliseconds.
- Over 16 GB of memcached.

The Architecture

- Ran into very public scaling problems. The little bird of failure popped up a lot for a while.
- Originally they had no monitoring, no graphs, no statistics, which makes it hard to pinpoint and solve problems. Added Munin and Nagios. There were difficulties using tools on Solaris. Had Google analytics but the pages weren't loading so it wasn't that

helpful :-)

- Use caching with memcached a lot.
 - For example, if getting a count is slow, you can memoize the count into memcache in a millisecond.
 - Getting your friends status is complicated. There are security and other issues. So rather than doing a query, a friend's status is updated in cache instead. It never touches the database. This gives a predictable response time frame (upper bound 20 msec).
 - ActiveRecord objects are huge so that's why they aren't cached. So they want to store critical attributes in a hash and lazy load the other attributes on access.
 - 90% of requests are API requests. So don't do any page/fragment caching on the front-end. The pages are so time sensitive it doesn't do any good. But they cache API requests.
- **Messaging**
 - Use message a lot. Producers produce messages, which are queued, and then are distributed to consumers. Twitter's main functionality is to act as a messaging bridge between different formats (SMS, web, IM, etc).
 - Send message to invalidate friend's cache in the background instead of doing all individually, synchronously.
 - Started with **DRb**, which stands for distributed Ruby. A library that allows you to send and receive messages from remote Ruby objects via TCP/IP. But it was a little flaky and single point of failure.
 - Moved to **Rinda**, which is a shared queue that uses a tuplespace model, along the lines of Linda. But the queues are persistent and the messages are lost on failure.
 - Tried Erlang. Problem: How do you get a broken server running at Sunday Monday with 20,000 users waiting? The developer didn't know. Not a lot of documentation. So it violates the use what you know rule.

- Moved to Starling, a distributed queue written in Ruby.
- Distributed queues were made to survive system crashes by writing them to disk.

Other big websites take this simple approach as well.

- SMS is handled using an API supplied by third party gateway's. It's very expensive.
- Deployment

- They do a review and push out new mongrel servers. No graceful way yet.
- An internal server error is given to the user if their mongrel server is replaced.
- All servers are killed at once. A rolling blackout isn't used because the message queue state is in the mongrels and a rolling approach would cause all the queues in the remaining mongrels to fill up.

- Abuse

- A lot of down time because people crawl the site and add everyone as friends. 9000 friends in 24 hours. It would take down the site.
- Build tools to detect these problems so you can pinpoint when and where they are happening.
- Be ruthless. Delete them as users.

- Partitioning

- Plan to partition in the future. Currently they don't. These changes have been enough so far.
- The partition scheme will be based on time, not users, because most requests are very temporally local.
- Partitioning will be difficult because of automatic **memoization**. They can't guarantee read-only operations will really be read-only. May write to a read-only slave, which is really bad.

- Twitter's API Traffic is 10x Twitter's Site

- Their API is the most important thing Twitter has done.

- Keeping the service simple allowed developers to build on top of their infrastructure and come up with ideas that are way better than Twitter could come up with. For example, *Twitterrific*, which is a beautiful way to use Twitter that a small team with different priorities could create.

- Monit is used to kill process if they get too big.

Lessons Learned

- Talk to the community. Don't hide and try to solve all problems yourself. Many brilliant people are willing to help if you ask.

- Treat your scaling plan like a business plan. Assemble a board of advisers to help you.

- Build it yourself. Twitter spent a lot of time trying other people's solutions that just almost seemed to work, but not quite. It's better to build some things yourself so you at least have some control and you can build in the features you need.

- Build in user limits. People will try to bust your system. Put in reasonable limits and detection mechanisms to protect your system from being killed.

- Don't make the database the central bottleneck of doom. Not everything needs to require a gigantic join. Cache data. Think of other creative ways to get the same result.

A good example is talked about in [Twitter, Rails, Hammers, and 11,000 Nails per Second](#).

- Make your application easily partitionable from the start. Then you always have a way to scale your system.

- Realize your site is slow. Immediately add reporting to track problems.

- Optimize the database.

- Index everything. Rails won't do this for you.

- Use explain to how your queries are running. Indexes may not be being as you expect.

- Denormalize a lot. Single handedly saved them. For example, they store all a user IDs friend IDs together, which prevented a lot of costly joins.

- Avoid complex joins.
- Avoid scanning large sets of data.
- Cache the hell out of everything. Individual active records are not cached, yet. The queries are fast enough for now.
- Test everything.
- You want to know when you deploy an application that it will render correctly.
- They have a full test suite now. So when the caching broke they were able to find the problem before going live.
- Long running processes should be abstracted to daemons.
- Use exception notifier and exception logger to get immediate notification of problems so you can address the right away.
- Don't do stupid things.
- Scale changes what can be stupid.
- Trying to load 3000 friends at once into memory can bring a server down, but when there were only 4 friends it works great.
- Most performance comes not from the language, but from application design.
- Turn your website into an open service by creating an API. Their API is a huge reason for Twitter's success. It allows user's to create an ever expanding and ecosystem around Twitter that is difficult to compete with. You can never do all the work your user's can do and you probably won't be as creative. So open you application up and make it easy for others to integrate your application with theirs.

Related Articles

- For a discussion of partitioning take a look at [Amazon Architecture](#), [An Unorthodox Approach to Database Design](#) : The Coming of the [Shard](#), [Flickr Architecture](#)
- The [Mailinator Architecture](#) has good strategies for abuse protection.

- [GoogleTalk Architecture](#) addresses some interesting issues when scaling social networking sites.
- [Example](#)
- [Memcached](#)
- [RoR](#)
- [Visit Scaling Twitter: Making Twitter 10000 Percent Faster](#)
- 26585 reads

Comments

Thu, 09/13/2007 - 22:51 — [Royans](#) (not verified)

Re: Scaling Twitter: Making Twitter 10000 Percent Faster

Todd, thanks for the excellent research u did on twitter. Its amazing that the entire Twitter infrastructure is running with just one rw database. Would be interesting to find out the usage stats on that single box...

- [reply](#)

Fri, 09/14/2007 - 00:15 — [Bob Warfield](#) (not verified)

Re: Scaling Twitter: Making Twitter 10000 Percent Faster

Loved your article, it echoes a lot of themes I've been talking about for awhile on my blog, so I wrote about the Twitter case based on your article here:

<http://smoothspan.wordpress.com/2007/09/14/twitter-scaling-story-mirrors...>

- [reply](#)

Sat, 09/15/2007 - 07:15 — [Shanti Braford](#) (not verified)

Re: Scaling Twitter: Making Twitter 10000 Percent Faster

I wonder what the **RoR** haters will make up now to say that ruby doesn't scale.

They loved jumping on the ruby hate bandwagon when twitter was going through it's difficulties. Little bo beep has been quite silent since.

Caching was the answer? Shock. Gasp. Awe. Just like **PHP**?!? Crazy!

- [reply](#)

Sat, 09/15/2007 - 11:23 — [Dave Hoover](#) (not verified)

Re: Scaling Twitter: Making Twitter 10000 Percent Faster

I think you're referring to [Starfish](#), not Starling.

Great article!

- [reply](#)

Thu, 09/20/2007 - 08:36 — [choonkeat](#) (not verified)

Re: Scaling Twitter: Making Twitter 10000 Percent Faster

No, its not Starfish. In the video of his presentation, he mentions "so I wrote Starling..."

- [reply](#)

Thu, 09/20/2007 - 16:02 — [miles](#) (not verified)

Re: Scaling Twitter: Making Twitter 10000 Percent Faster

great article (and site) Todd. thanks for pulling all this information together. It's a great resource

ps. @Dave: Blaine referred to his 'starling' messaging framework at the SJ Ruby Conference earlier in the year.

- [reply](#)

Sat, 09/22/2007 - 14:01 — [Marcus](#) (not verified)

They could have been 20% better?

So, let's be clear, the biased source in defense mode says themselves they could have been 20% faster just by selecting a different language (note that it doesn't exactly say what the performance hit of the Rails framework itself is, so let's just go with 20% improvement by changing languages and ignore potential problems in (1) their coding decisions and (2) their chosen framework).... Wow, sign me up for an easy 20% improvement!

Yeah, yeah, I know, I'll hear the usual tripe about how amazing fast Ruby is to develop with. Visual Basic is pretty easy too, as is **PHP**, but I don't use those either.

- [reply](#)

Mon, 09/24/2007 - 08:02 — Mikael (not verified)

Re: Scaling Twitter: Making Twitter 10000 Percent Faster

Sounds like **Ruby on Rails** _was_ to blame as the 10000 percent improvement was reached by essentially removing the "on rails" part of the equation by extensive caching. This seems to be the real weakness of **RoR**; Ruby in itself seems OK performance-wise, slower than **PHP** for example but not catastrophically so. **PHP** is slower than **Java** but scales nicely anyway. The database abstraction in "on rails" is a real performance killer though and all the high traffic sites that use RoR successfully (twitter, penny arcade, ...) seems to have taken steps to avoid using the database abstraction on live page views by extensive caching.

Of course, caching is a necessary tool for scaling regardless of the platform but with a less inefficient abstraction layer than the one in RoR it is possible to grow more before you have to recode stuff for caching.

- [reply](#)

Fri, 10/12/2007 - 18:07 — [Dustin Puryear](#) (not verified)

Re: Scaling Twitter: Making Twitter 10000 Percent Faster

Excellent article.

I agree with one of the other commenters that it's surprising they have this running from a single **MySQL** server. Wow. The fact that twitter tends to be very write-heavy, and MySQL isn't exactly perfect for multimaster replication architectures probably has a lot to do with that. I wonder what they are planning to do for future growth? Obviously this will not continue to work as-is..

--

Dustin Puryear

Author, "Best Practices for Managing **Linux** and UNIX Servers"

<http://www.puryear-it.com/pubs/linux-unix-best-practices>

I Google Architecture



Mon, 07/23/2007 - 04:26 — [Todd Hoff](#)

- [Google Architecture \(1526\)](#)

Google is the King of scalability. Everyone knows Google for their large, sophisticated, and fast searching, but they don't just shine in search. Their platform approach to building scalable applications allows them to roll out internet scale applications at an alarmingly high competition crushing rate. Their goal is always to build a higher performing higher scaling infrastructure to support their products. How do they do that?

Information Sources

- Video: Building Large Systems at Google
- Google Lab: The Google File System
- Google Lab: MapReduce: Simplified Data Processing on Large Clusters
- Google Lab: BigTable.
- Video: BigTable: A Distributed Structured Storage System.
- Google Lab: The Chubby Lock Service for Loosely-Coupled Distributed Systems.
- How Google Works by David Carr in Baseline Magazine.
- Google Lab: Interpreting the Data: Parallel Analysis with Sawzall.
- Dare Obasonjo's Notes on the scalability conference.

Platform

- Linux
- A large diversity of languages: Python, Java, C++

What's Inside?

The Stats

- Estimated 450,000 low-cost commodity servers in 2006
- In 2005 Google indexed 8 billion web pages. By now, who knows?
- Currently there over 200 GFS clusters at Google. A cluster can have 1000 or even 5000 machines. Pools of tens of thousands of machines retrieve data from GFS clusters that run as large as 5 petabytes of storage. Aggregate read/write throughput can be as high as 40 gigabytes/second across the cluster.
- Currently there are 6000 MapReduce applications at Google and hundreds of new applications are being written each month.

- BigTable scales to store billions of URLs, hundreds of terabytes of satellite imagery, and preferences for hundreds of millions of users.

The Stack

Google visualizes their infrastructure as a three layer stack:

- Products: search, advertising, **email**, maps, video, chat, blogger
- Distributed Systems Infrastructure: GFS, MapReduce, and BigTable.
- Computing Platforms: a bunch of machines in a bunch of different data centers
- Make sure easy for folks in the company to deploy at a low cost.
- Look at price performance data on a per application basis. Spend more money on hardware to not lose log data, but spend less on other types of data. Having said that, they don't lose data.

Reliable Storage Mechanism with GFS (Google File System)

- Reliable scalable storage is a core need of any application. GFS is their core storage platform.
- Google File System - large distributed log structured file system in which they throw in a lot of data.
- Why build it instead of using something off the shelf? Because they control everything and it's the platform that distinguishes them from everyone else. They required:
 - high reliability across data centers
 - scalability to thousands of network nodes
 - huge read/write bandwidth requirements
 - support for large blocks of data which are gigabytes in size.
 - efficient distribution of operations across nodes to reduce bottlenecks
- System has master and chunk servers.
 - **Master** servers keep metadata on the various data files. Data are stored in the file

system in 64MB chunks. Clients talk to the master servers to perform metadata operations on files and to locate the chunk server that contains the needed they need on disk.

- Chunk servers store the actual data on disk. Each chunk is replicated across three different chunk servers to create redundancy in case of server crashes. Once directed by a master server, a client application retrieves files directly from chunk servers.

- A new application coming on line can use an existing GFS cluster or they can make your own. It would be interesting to understand the provisioning process they use across their data centers.

- Key is enough infrastructure to make sure people have choices for their application. GFS can be tuned to fit individual application needs.

Do Something With the Data Using MapReduce

- Now that you have a good storage system, how do you do anything with so much data? Let's say you have many TBs of data stored across a 1000 machines. Databases don't scale or cost effectively scale to those levels. That's where MapReduce comes in.

- 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. 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.

- Why use MapReduce?
 - Nice way to partition tasks across lots of machines.
 - Handle machine failure.
 - Works across different application types, like search and ads. Almost every application has map reduce type operations. You can precompute useful data, find word counts, sort TBs of data, etc.
 - Computation can automatically move closer to the IO source.
- The MapReduce system has three different types of servers.
 - The Master server assigns user tasks to map and reduce servers. It also tracks the state of the tasks.
 - The Map servers accept user input and performs map operations on them. The results are written to intermediate files
 - The Reduce servers accepts intermediate files produced by map servers and performs reduce operation on them.
- For example, you want to count the number of words in all web pages. You would feed all the pages stored on GFS into MapReduce. This would all be happening on 1000s of machines simultaneously and all the coordination, job scheduling, failure handling, and data transport would be done automatically.
 - The steps look like: GFS -> Map -> Shuffle -> Reduction -> Store Results back into GFS.
 - In MapReduce a map maps one view of data to another, producing a key value pair, which in our example is word and count.
 - Shuffling aggregates key types.
 - The reductions sums up all the key value pairs and produces the final answer.
- The Google indexing pipeline has about 20 different map reductions. A pipeline looks at data with a whole bunch of records and aggregating keys. A second map-reduce comes a long, takes that result and does something else. And so on.

- Programs can be very small. As little as 20 to 50 lines of code.
- One problem is stragglers. A straggler is a computation that is going slower than others which holds up everyone. Stragglers may happen because of slow IO (say a bad controller) or from a temporary CPU spike. The solution is to run multiple of the same computations and when one is done kill all the rest.
- Data transferred between map and reduce servers is compressed. The idea is that because servers aren't CPU bound it makes sense to spend on data compression and decompression in order to save on bandwidth and I/O.

Storing Structured Data in BigTable

- BigTable is a large scale, fault tolerant, self managing system that includes terabytes of memory and petabytes of storage. It can handle millions of reads/writes per second.
- BigTable is a distributed hash mechanism built on top of GFS. It is not a relational database. It doesn't support joins or SQL type queries.
- It provides lookup mechanism to access structured data by key. GFS stores opaque data and many applications needs has data with structure.
- Commercial databases simply don't scale to this level and they don't work across 1000s machines.
- By controlling their own low level storage system Google gets more control and leverage to improve their system. For example, if they want features that make cross data center operations easier, they can build it in.
- Machines can be added and deleted while the system is running and the whole system just works.
- Each data item is stored in a cell which can be accessed using a row key, column key, or timestamp.
- Each row is stored in one or more tablets. A tablet is a sequence of 64KB blocks in a data format called SSTable.

- BigTable has three different types of servers:
 - The Master servers assign tablets to tablet servers. They track where tablets are located and redistributes tasks as needed.
 - The Tablet servers process read/write requests for tablets. They split tablets when they exceed size limits (usually 100MB - 200MB). When a tablet server fails, then a 100 tablet servers each pickup 1 new tablet and the system recovers.
 - The Lock servers form a distributed lock service. Operations like opening a tablet for writing, Master arbitration, and access control checking require mutual exclusion.
- A locality group can be used to physically store related bits of data together for better locality of reference.
- Tablets are cached in RAM as much as possible.

Hardware

- When you have a lot of machines how do you build them to be cost efficient and use power efficiently?
- Use ultra cheap commodity hardware and built software on top to handle their death.
- A 1,000-fold computer power increase can be had for a 33 times lower cost if you use a failure-prone infrastructure rather than an infrastructure built on highly reliable components. You must build reliability on top of unreliability for this strategy to work.
- Linux, in-house rack design, PC class mother boards, low end storage.
- Price per wattage on performance basis isn't getting better. Have huge power and cooling issues.
- Use a mix of collocation and their own data centers.

Misc

- Push changes out quickly rather than wait for QA.
- Libraries are the predominant way of building programs.
- Some applications are provided as services, like crawling.

- An infrastructure handles versioning of applications so they can be release without a fear of breaking things.

Future Directions for Google

- Support geo-distributed clusters.
- Create a single global namespace for all data. Currently data is segregated by cluster.
- More and better automated migration of data and computation.
- Solve consistency issues that happen when you couple wide area replication with network partitioning (e.g. keeping services up even if a cluster goes offline for maintenance or due to some sort of outage).

Lessons Learned

- Infrastructure can be a competitive advantage. It certainly is for Google. They can roll out new internet services faster, cheaper, and at scale at few others can compete with. Many companies take a completely different approach. Many companies treat infrastructure as an expense. Each group will use completely different technologies and their will be little planning and commonality of how to build systems. Google thinks of themselves as a systems engineering company, which is a very refreshing way to look at building software.
- Spanning multiple data centers is still an unsolved problem. Most websites are in one and at most two data centers. How to fully distribute a website across a set of data centers is, shall we say, tricky.
- Take a look at **Hadoop** (product) if you don't have the time to rebuild all this infrastructure from scratch yourself. Hadoop is an open source implementation of many of the same ideas presented here.
- An under appreciated advantage of a platform approach is junior developers can quickly and confidently create robust applications on top of the platform. If every project

needs to create the same distributed infrastructure wheel you'll run into difficulty because the people who know how to do this are relatively rare.

- Synergy isn't always crap. By making all parts of a system work together an improvement in one helps them all. Improve the file system and everyone benefits immediately and transparently. If every project uses a different file system then there's no continual incremental improvement across the entire stack.
- Build self-managing systems that work without having to take the system down. This allows you to more easily rebalance resources across servers, add more **capacity** dynamically, bring machines off line, and gracefully handle upgrades.
- Create a Darwinian infrastructure. Perform time consuming operation in parallel and take the winner.
- Don't ignore the Academy. Academia has a lot of good ideas that don't get translated into production environments. Most of what Google has done has prior art, just not prior large scale **deployment**.
- Consider compression. Compression is a good option when you have a lot of CPU to throw around and limited IO.

I Digg Architecture



Tue, 08/07/2007 - 01:28 — [Todd Hoff](#)

- [Digg Architecture \(966\)](#)

Traffic generated by Digg's over 1.2 million famously info-hungry users can crash an unsuspecting website head-on into its CPU, memory, and bandwidth limits. How does Digg handle all this load?

Site: <http://digg.com>

Information Sources

- [How Digg.com uses the LAMP](#) stack to scale upward
- [Digg PHP's Scalability and Performance](#)

Platform

- [MySQL](#)
- [Linux](#)
- [PHP](#)
- [Lucene](#)
- [APC PHP Accelerator](#)
- [MCache](#)

The Stats

- Started in late 2004 with a single Linux server running [Apache](#) 1.3, PHP 4, and MySQL. 4.0 using the default MyISAM storage engine
- Over 1.2 million users.
- Over 200 million page views per month
- 100 servers hosted in multiple data centers.
 - 20 database servers
 - 30 Web servers
 - A few search servers running Lucene.
 - The rest are used for redundancy.
- 30GB of data.
- None of the scaling challenges we faced had anything to do with PHP. The biggest

issues faced were database related.

- The lightweight nature of PHP allowed them to move processing tasks from the database to PHP in order to improve scaling. Ebay does this in a radical way. They moved nearly all work out of the database and into applications, including joins, an operation we normally think of as the job of the database.

What's Inside

- Load balancer in the front that sends queries to PHP servers.
- Uses a MySQL master-slave setup.
- Transaction-heavy servers use the InnoDB storage engine.
- OLAP-heavy servers use the MyISAM storage engine.
- They did not notice a performance degradation moving from MySQL 4.1 to version 5.
- **Memcached** is used for caching.
- Sharding is used to break the database into several smaller ones.
- Digg's usage pattern makes it easier for them to scale. Most people just view the front page and leave. Thus 98% of Digg's database accesses are reads. With this balance of operations they don't have to worry about the complex work of architecting for writes, which makes it a lot easier for them to scale.
- They had problems with their storage system telling them writes were on disk when they really weren't. Controllers do this to improve the appearance of their performance. But what it does is leave a giant data integrity whole in failure scenarios. This is really a pretty common problem and can be hard to fix, depending on your hardware setup.
- To lighten their database load they used the APC PHP accelerator MCache.
- You can configure PHP not parse and compile on each load using a combination of Apache 2's worker threads, FastCGI, and a PHP accelerator. On a page's first load the PHP code is compiled so any subsequent page loads are very fast.

Lessons Learned

- Tune MySQL through your database engine selection. Use InnoDB when you need transactions and MyISAM when you don't. For example, transactional tables on the master can use MyISAM for read-only slaves.
- At some point in their growth curve they were unable to grow by adding RAM so had to grow through architecture.
- People often complain Digg is slow. This is perhaps due to their large javascript libraries rather than their backend architecture.
- One way they scale is by being careful of which application they deploy on their system. They are careful not to release applications which use too much CPU. Clearly Digg has a pretty standard LAMP architecture, but I thought this was an interesting point. Engineers often have a bunch of cool features they want to release, but those features can kill an infrastructure if that infrastructure doesn't grow along with the features. So push back until your system can handle the new features. This goes to **capacity** planning, something the Flickr emphasizes in their scaling process.
- You have to wonder if by limiting new features to match their infrastructure might Digg lose ground to other faster moving social bookmarking services? Perhaps if the infrastructure was more easily scaled they could add features faster which would help them compete better? On the other hand, just adding features because you can doesn't make a lot of sense either.
- The data layer is where most scaling and performance problems are to be found and these are language specific. You'll hit them using **Java**, PHP, Ruby, or insert your favorite language here.

An Unorthodox Approach to Database Design : The Coming of the Shard



Tue, 07/31/2007 - 18:13 — [Todd Hoff](#)

- [An Unorthodox Approach to Database Design : The Coming of the Shard \(1136\)](#)

Once upon a time we scaled databases by buying ever bigger, faster, and more expensive machines. While this arrangement is great for big iron profit margins, it doesn't work so well for the bank accounts of our heroic system builders who need to scale well past what they can afford to spend on giant database servers. In a extraordinary two article series, Dathan Pattishall, explains his motivation for a revolutionary new database architecture--sharding--that he began thinking about even before he worked at Friendster, and fully implemented at Flickr. Flickr now handles more than 1 billion transactions per day, responding in less then a few seconds and can scale linearly at a low cost.

What is sharding and how has it come to be the answer to large website scaling problems?

Information Sources

- * [Unorthodox approach to database design Part1:History](#)
- * [Unorthodox approach to database design Part 2:Friendster](#)

What is sharding?

While working at Auction Watch, Dathan got the idea to solve their scaling problems by creating a database server for a group of users and running those servers on cheap **Linux** boxes. In this scheme the data for User A is stored on one server and the data for User B is stored on another server. It's a federated model. Groups of 500K users are stored together in what are called *shards*.

The advantages are:

- **High availability.** If one box goes down the others still operate.
- **Faster queries.** Smaller amounts of data in each user group mean faster querying.
- **More write bandwidth.** With no master database serializing writes you can write in parallel which increases your write throughput. Writing is major bottleneck for many websites.
- **You can do more work.** A parallel backend means you can do more work simultaneously. You can handle higher user loads, especially when writing data, because there are parallel paths through your system. You can load balance web servers, which access shards over different network paths, which are processed by separate CPUs, which use separate caches of RAM and separate disk IO paths to process work. Very few bottlenecks limit your work.

I How is sharding different than traditional architectures?

Sharding is different than traditional database architecture in several important ways:

- **Data are denormalized.** Traditionally we normalize data. Data are splayed out into anomaly-less tables and then joined back together again when they need to be used. In sharding the data are denormalized. You store together data that are used together.

This doesn't mean you don't also segregate data by type. You can keep a user's profile data separate from their comments, blogs, **email**, media, etc, but the user profile data would be stored and retrieved as a whole. This is a very fast approach. You just get a blob and store a blob. No joins are needed and it can be written with one disk write.

- **Data are parallelized across many physical instances.** Historically database servers are scaled up. You buy bigger machines to get more power. With sharding the data are parallelized and you scale by scaling out. Using this approach you can get massively more work done because it can be done in parallel.
- **Data are kept small.** The larger a set of data a server handles the harder it is to cash intelligently because you have such a wide diversity of data being accessed. You need huge gobs of RAM that may not even be enough to cache the data when you need it. By isolating data into smaller shards the data you are accessing is more likely to stay in cache.

Smaller sets of data are also easier to backup, restore, and manage.

- **Data are more highly available.** Since the shards are independent a failure in one doesn't cause a failure in another. And if you make each shard operate at 50% **capacity** it's much easier to upgrade a shard in place. Keeping multiple data copies within a shard also helps with redundancy and making the data more parallelized so more work can be done on the data. You can also setup a shard to have a master-slave or dual master relationship within the shard to avoid a single point of failure within the shard. If one server goes down the other can take over.
- **It doesn't use replication.** Replicating data from a master server to slave servers is a traditional approach to scaling. Data is written to a master server and then replicated to one or more slave servers. At that point read operations can be handled by the slaves, but all writes happen on the master.

Obviously the master becomes the write bottleneck and a single point of failure. And as load increases the cost of replication increases. Replication costs in CPU, network bandwidth, and disk IO. The slaves fall behind and have stale data. The folks at [YouTube](#) had a big problem with replication overhead as they scaled.

Sharding cleanly and elegantly solves the problems with replication.

Some **Problems** With Sharding

Sharding isn't perfect. It does have a few problems.

- **Rebalancing data.** What happens when a shard outgrows your storage and needs to be split? Let's say some user has a particularly large friends list that blows your storage capacity for the shard. You need to move the user to a different shard.

On some platforms I've worked on this is a killer problem. You had to build out the data center correctly from the start because moving data from shard to shard required a lot of downtime.

Rebalancing has to be built in from the start. Google's shards automatically rebalance. For this to work data references must go through some sort of naming service so they can be relocated. This is what Flickr does. And your references must be invalidateable so the underlying data can be moved while you are using it.

- **Joining data from multiple shards.** To create a complex friends page, or a user profile page, or a thread discussion page, you usually must pull together lots of different data from many different sources. With sharding you can't just issue a query and get back all the data. You have to make individual requests to your data sources, get all the responses, and then build the page. Thankfully, because of caching and fast networks this process is usually fast enough that your page load times can be excellent.
- **How do you partition your data in shards?** What data do you put in which shard?

Where do comments go? Should all user data really go together, or just their profile data? Should a user's media, IMs, friends lists, etc go somewhere else? Unfortunately there are no easy answer to these questions.

- **Less leverage.** People have experience with traditional RDBMS tools so there is a lot of help out there. You have books, experts, tool chains, and discussion forums when something goes wrong or you are wondering how to implement a new feature. Eclipse won't have a shard view and you won't find any automated backup and restore programs for your shard. With sharding you are on your own.
- **Implementing shards is not well supported.** Sharding is currently mostly a roll your own approach. [LiveJournal](#) makes their tool chain available. Hibernate has a [library](#) under development. [MySQL](#) has added support for [partitioning](#). But in general it's still something you must implement yourself.

Comments

Tue, 07/31/2007 - 22:12 — [Tim](#) (not verified)

Great post !

This is probably the most interesting post I've read in a long long time. Thanks for sharing the advantages and drawbacks of sharding.... and thanks for putting together all these resources/info about scaling... it's really really interesting.

- [reply](#)

Wed, 08/01/2007 - 20:22 — [Vinit](#) (not verified)

Thanks for this info.

Thanks for this info. Helped me understand about what the heck to do with all this user data coming my way!!!

- [reply](#)

Wed, 08/01/2007 - 23:17 — [Ryan T Mulligan](#) (not verified)

Intranet?

I dislike how your link of livejournal does not actually go to a livejournal website, or information about their toolchain.

- [reply](#)

Thu, 08/02/2007 - 01:48 — [Todd Hoff](#)



re: intranet

I am not sure what you mean about live journal. It goes to a page on this site which references two danga.com sites. Oh I see, memcached goes to a category link which doesn't include memcached. The hover text does include the link, but I'll add it in. Good catch. Thanks.

- [reply](#)

Thu, 08/02/2007 - 15:10 — [tim wee](#) (not verified)

Question about a statement in the post

"Sharding cleanly and elegantly solves the problems with replication."

Is this true? You do need to replicate still right? You need duplication and a copy of the data that is not too stale in case one of your shards go down? So you still need to replicate correct?

- [reply](#)

Thu, 08/02/2007 - 15:21 — [Todd Hoff](#)



sharding and replication

> Is this true? You do need to replicate still right?

You won't have the problems with replication overhead and lag because you are writing to a appropriately sized shard rather than a single master that must replicate to all its slaves, assuming you have a lot of slaves.

You will still replicate within the shard, but that will be more of a fixed reasonable cost because the number of slaves will be small.

Google replicates content 3 times, so even in that case it's more of a fixed overhead then chaining a lot of slaves together.

That's my understanding at least.

- [reply](#)

Sat, 08/04/2007 - 17:23 — [Kevin Burton](#) (not verified)

We might OSS our sharding framework

We've been building out a sharding framework for use with Spinn3r. It's about to be deployed into production this week.

We're very happy with the way things have moved forward and will probably be OSSing it.

We aggregate a LOT of data on behalf of our customers so have huge data storage requirements.

Kevin

- [reply](#)

Sat, 08/04/2007 - 21:53 — [Anonymous](#) (not verified)

Lookup table

Do you still need a master lookup table? How do you know which shard has the data you need to access?

- [reply](#)

Sat, 08/04/2007 - 23:01 — [Todd Hoff](#)



re: Lookup table

I think a lookup table is the most flexible option. It allows for flexible shard assignment algorithms and you can change the mapping when you need to. Here a few other ideas.

I am sure there are more.

[Flickr](#) talks about a "global ring" that is like DNS that maps a key to a shard ID. The map is kept in memcached for a 1/2 hour or so. I bought Cal Henderson's book and it should arrive soon. If he has more details I'll do a quick write up.

Users could be assigned to shards initially on a round robin basis or through some other whizzbang algorithm.

You could embed shard IDs into your assigned row IDs so the mapping is obvious from the ID.

You could hash on a key to a shard bucket.

You could partition based on on key values. So users with names starting with A-C go to shard1, that sort of thing. [MySQL](#) has a number of different partition rules as examples.

- [reply](#)

Mon, 08/06/2007 - 02:13 — Frank (not verified)

Thank you

It's helpful to know how the big players handle their scaling issues.

Thanks for sharing!

- [reply](#)

Thu, 08/09/2007 - 12:43 — Anonymous (not verified)

Sharding

Attacking sharding from the application layer is simply wrong. This functionality should be left to the DBMS. Access from the app layer would be transparent and it would be up to the DB admin to configure the data servers correctly for sharding to automatically and transparently scale out across them.

If you are implementing sharding from the app layer you are getting yourself in a very tight corner and one day will find out how stuck you are there. This is the cornerstone of improper delegation of the functionalities in a multi-tier system.

- [reply](#)

Mon, 08/13/2007 - 14:44 — Diogin (not verified)

>How do you partition your

>How do you partition your data in shards? What data do you put in which shard? Where do comments go? Should all user data really go together, or just their profile data?

Should a user's media, IMs, friends lists, etc go somewhere else? Unfortunately there are no easy answer to these questions.

I have exactly the question to ask..

I've referred the architectures of [LiveJournal](#) and Mixi, both of which introduce shards.

However, I saw a "Global Cluster" which store meta informations for other clusters. By doing this we get an extreme heavy cluster, it must handle all the cluster_id <-> user_id metas and lost the advantage of sharding...Is it?

The other way, partition by algorithms on keys, is difficult in transition.

So, could you give me some advice? Thank you very much for sharing experiences :)

- [reply](#)

Mon, 08/13/2007 - 15:36 — [Todd Hoff](#)



> By doing this we get an

> By doing this we get an extreme heavy cluster, it must handle

> all the cluster_id <-> user_id metas

I think the idea is that because these mapping is so small they can all be cached in RAM and thus their resolution can be very very fast. Do you think that would be too slow?

- [reply](#)

Mon, 08/13/2007 - 17:11 — [Diogin](#) (not verified)

Yeah, you reminded me! I

Yeah, you reminded me! I have a little doubts on this before, when I think the table would be terribly huge as all the table records on other clusters are all gathered in this table, and all queries should first refer to this cluster.

Maybe I can use memcached clusters to cache them. Thank you :)

- [reply](#)

Tue, 08/14/2007 - 17:33 — [Arnon Rotem-Gal-Oz](#) (not verified)

Partitioning is the new standard

If you look at the architectures of all the major internet-scale sites (such as eBay, Amazon, Flickr etc.) you'd see they've come to the same conclusion and same patterns

I've also published an article on InfoQ discussing this topic yesterday (I only found this site now or I would have included it in the article)

Arnon

- [reply](#)

Wed, 08/22/2007 - 21:24 — [Todd Hoff](#)



More Good Info on Partitioning

Jeremy Cole and Eric Bergen have an excellent section on database partitioning starting on about [page 14](#) of [MySQL Scaling and High Availability Architectures](#). They talk about different partitioning models, difficulties with partitioning, HiveDB, Hibernate Shards, and server architectures.

I'll definitely do a write up on this document a little later, but if interested dive in now.

- [reply](#)

Fri, 09/14/2007 - 11:20 — [Norman Harebottle](#) (not verified)

Re: An Unorthodox Approach to Database Design : The Coming of th

I agree with the above post questioning the placement of partitioning logic in the application layer. Why not write the application layer against a logical model (NOT a

storage model!) and then just engineer the existing data storage abstraction mechanism (DBMS engine) such that it will handle the partitioning functionality in a parallel manner?

I would be very interested to see a study done comparing the architectures of this sharding concept against a federated database design such as what is described on this site http://www.sql-server-performance.com/tips/federated_databases_p1.aspx

- [reply](#)

Fri, 09/14/2007 - 15:22 — [Todd Hoff](#)



Re: is the logical model the correct place for partitioning?

> I would be very interested to see a study done comparing the

> architectures of this sharding concept against a federated database design

Most of us don't have accesses to a real affordable federated database (parallel queries), so it's somewhat a moot point :-). And even these haven't been able to scale at the highest levels anyway.

The advantage of partitioning in the application layer is that you are not bottlenecked on the traffic cop server that must analyze SQL and redistribute work to the proper federations and then integrate the results. You go directly to where the work needs to be done.

I understand the architectural drive to push this responsibility to a logical layer, but it's hard to ignore the advantages of using client side resources to go directly to the right shard based on local context. What is the cost? A call to library code that must be kept in sync with the partitioning scheme. Is that really all that bad?

- [reply](#)

Thu, 09/20/2007 - 14:10 — Anonymous (not verified)

Re: An Unorthodox Approach to Database Design : The Coming of th

Sorry, this isn't new, except maybe to younger programmers. I've been using this approach for many years. Federated databases with horizontally partitioned data is old news. Its just not taught as a standard technique for scaling, mostly because scaling isn't taught as a core subject. (Why is that, do you suppose? Too hard to cover? Practical examples not practical in an academic setting?)

The reason this is getting attention now is a perfect storm of cheap hardware, nearly free network connectivity, free database software, and beaucoup users (testers?) over the 'net.

- [reply](#)

Sat, 09/22/2007 - 13:53 — Marcus (not verified)

Useful, interesting, but not new

Great content and great site... except for the worshipful adoration of these young teams who seem to think they've each discovered America.

I could go on at length about Flickr's performance troubles and extremely slow changes after their early success, Twitter's meltdowns and indefensible defense of the slowest language in wide use on the net, and old-school examples of horizontal partitioning (AKA "sharding" heh), but I'll spare you. This cotton candy sugary sweet lovin' of Web 2.0 darlings really is a bit tiresome.

Big kudos though on deep coverage of the subject matter between the cultic chants. :-D

- [reply](#)

Sat, 09/22/2007 - 18:39 — [Todd Hoff](#)



Re: Useful , interesting, but not new

> Big kudos though on deep coverage of the subject matter between the cultic
chants. :-D

I am actually more into root music. But thanks, I think :-)

- [reply](#)

Fri, 09/28/2007 - 06:11 — [Sean Bannister](#) (not verified)

Re: An Unorthodox Approach to Database Design : The Coming of th

Good article, very interesting to read.

- [reply](#)

Sat, 09/29/2007 - 06:46 — [Ed](#) (not verified)

Re: An Unorthodox Approach to Database Design : The Coming of th

Fanball.com has been using this technique for its football commissioner product for
years. As someone else commented, it used to be called horizontal partitioning back
then. Does it cost less if it's called sharding? :-)

- [reply](#)

Mon, 10/08/2007 - 03:02 — Anonymous (not verified)

Re: An Unorthodox Approach to Database Design

Why is this even news, we did something similar in my old job. Split up different clients
among different server stacks. Move along nothing to see...

- [reply](#)

Sun, 10/14/2007 - 11:29 — Anonymous (not verified)

Re: An Unorthodox Approach to Database Design : The Coming of th

Could it be the fact that people are pulling this off with mysql and berkeleydb that is making horizontal partitioning interesting? When you compare two solutions, one using an open source database and one using a closed source database, is one solution more inherently scalable? Well all things being equal performance wise, its nice to not have to do a purchase order for the closed source software, so I would say that is why this is getting all the 'hype'. Old school oracle/mssqlserver patronizing DBAs are getting schooled by non-dbas who are setting up the *highest* data throughput architectures and not using sql server or oracle. That is why this is getting high visibility.

Believe or not many people still say mysql/berkeleydb is a toy outside of some of the major tech hubs. Stories like this are what make people, especially dbas, listen. The only recourse is 'I have done that before with xxx database'. Well you should be the one that suggests doing it with the open source 'toy' database then, if you are so good.

In my experience there are many old-school DBAs that are in denial that this kind of architecture is capable of out performing their *multi-million dollar oracle software purchase decisions* and they don't want to admit it.

- [reply](#)

Wed, 10/24/2007 - 13:12 — [Harel Malka](#) (not verified)

Re: An Unorthodox Approach to Database Design : The Coming of th

What I'm most interested in relating to Shards is people's thoughts and experience in migrating TO a shard approach from a single database, and moving (large amounts of) data around from shard to shard. In particular - strategies to maintain referential integrity as we're moving data by a user.

As well, should you need to query data joining user A and user B which both reside on different shards - what approaches people see as fit?

Harel

I LiveJournal Architecture



Mon, 07/09/2007 - 16:57 — Todd Hoff

- [LiveJournal Architecture \(608\)](#)

A fascinating and detailed story of how **LiveJournal** evolved their system to scale.

LiveJournal was an early player in the free blog service race and faced issues from quickly adding a large number of users. Blog posts come fast and furious which causes a lot of writes and writes are particularly hard to scale. Understanding how LiveJournal faced their scaling problems will help any aspiring website builder.

Site: <http://www.livejournal.com/>

Information Sources

- [LiveJournal](#) - Behind The Scenes Scaling Storytime
- [Google Video](#)
- [Tokyo Video](#)
- [2005 version](#)

Platform

- **Linux**
- **MySql**

- Perl
- Memcached
- MogileFS
- Apache

What's Inside?

- Scaling from 1, 2, and 4 hosts to cluster of servers.
- Avoid single points of failure.
- Using MySQL replication only takes you so far.
- Becoming IO bound kills scaling.
- Spread out writes and reads for more parallelism.
- You can't keep adding read slaves and scale.
- Shard storage approach, using DRBD, for maximal throughput. Allocate shards based on roles.
- Caching to improve performance with memcached. Two-level hashing to distributed RAM.
- Perlbal for web load balancing.
- MogileFS, a distributed file system, for parallelism.
- TheSchwartz and Gearman for distributed job queuing to do more work in parallel.
- Solving persistent connection problems.

Lessons Learned

- Don't be afraid to write your own software to solve your own problems. LiveJournal as provided incredible value to the community through their efforts.
- Sites can evolve from small 1, 2 machine setups to larger systems as they learn about their users and what their system really needs to do.

- Parallelization is key to scaling. Remove choke points by caching, load balancing, sharding, clustering file systems, and making use of more disk spindles.
- Replication has a cost. You can't just keep adding more and more read slaves and expect to scale.
- Low level issues like which OS event notification mechanism to use, file system and disk interactions, threading and even models, and connection types, matter at scale.
- Large sites eventually turn to a distributed queuing and scheduling mechanism to distribute large work loads across a grid.

I GoogleTalk Architecture



Mon, 07/23/2007 - 22:47 — Todd Hoff

- [GoogleTalk Architecture \(549\)](#)

Google Talk is Google's instant communications service. Interestingly the IM messages aren't the major architectural challenge, handling user presence indications dominate the design. They also have the challenge of handling small low latency messages and integrating with many other systems. How do they do it?

Site: <http://www.google.com/talk>

Information Sources

- [GoogleTalk Architecture](#)

Platform

- [Linux](#)

- Java
- Google Stack
- Shard

What's Inside?

The Stats

- Support presence and messages for millions of users.
- Handles billions of packets per day in under 100ms.
- IM is different than many other applications because the requests are small packets.
- Routing and application logic are applied per packet for sender and receiver.
- Messages must be delivered in-order.
- Architecture extends to new clients and Google services.

Lessons Learned

- Measure the right thing.
 - People ask about how many IMs do you deliver or how many active users. Turns out not to be the right engineering question.
 - Hard part of IM is how to show correct present to all connected users because growth is non-linear: $\text{ConnectedUsers} * \text{BuddyListSize} * \text{OnlineStateChanges}$
 - A linear user grown can mean a very non-linear server growth which requires serving many billions of presence packets per day.
 - Have a large number friends and presence explodes. The number IMs not that big of deal.
- Real Life Load Tests
 - Lab tests are good, but don't tell you enough.
 - Did a backend launch before the real product launch.

- Simulate presence requests and going on-line and off-line for weeks and months, even if real data is not returned. It works out many of the kinks in network, failover, etc.

- Dynamic Resharding

- Divide user data or load across shards.
- Google Talk backend servers handle traffic for a subset of users.
- Make it easy to change the number of shards with zero downtime.
- Don't shard across data centers. Try and keep users local.
- Servers can bring down servers and backups take over. Then you can bring up new servers and data migrated automatically and clients auto detect and go to new servers.

- Add Abstractions to Hide System Complexity

- Different systems should have little knowledge of each other, especially when separate groups are working together.
- Gmail and Orkut don't know about sharding, load-balancing, or fail-over, data center architecture, or number of servers. Can change at anytime without cascading changes throughout the system.
- Abstract these complexities into a set of gateways that are discovered at runtime.
- RPC infrastructure should handle rerouting.

- Understand Semantics of Lower Level Libraries

- Everything is abstracted, but you must still have enough knowledge of how they work to architect your system.
- Does your RPC create TCP connections to all or some of your servers? Very different implications.
- Does the library performance health checking? This is architectural implications as you can have separate system failing independently.
- Which kernel operation should you use? IM requires a lot connections but few have any

activity. Use **epoll** vs poll/select.

- Protect Against Operation **Problems**

- Smooth out all spikes in server activity graphs.
- What happens when servers restart with an empty cache?
- What happens if traffic shifts to a new data center?
- Limit cascading problems. Back off from busy servers. Don't accept work when sick.
- Isolate in emergencies. Don't infect others with your problems.
- Have intelligent retry logic policies abstracted away. Don't sit in hard 1msec retry loops, for example.

- Any Scalable System is a Distributed System

- Add fault tolerance to every component of the system. Everything fails.
- Add ability to profile live servers without impacting server. Allows continual improvement.
- Collect metrics from server for monitoring. Log everything about your system so you see patterns in cause and effects.
- Log end-to-end so you can reconstruct an entire operation from beginning to end across all machines.

- Software Development Strategies

- Make sure binaries are both backward and forward compatible so you can have old clients work with new code.
- Build an experimentation framework to try new features.
- Give engineers access to product machines. Gives end-to-end ownership. This is very different than many companies who have completely separate OP teams in their data centers. Often developers can't touch production machines.