NoSQL数据库笔谈

databases , appdir , node , paper
颜开 , v0.2 , 2010.2

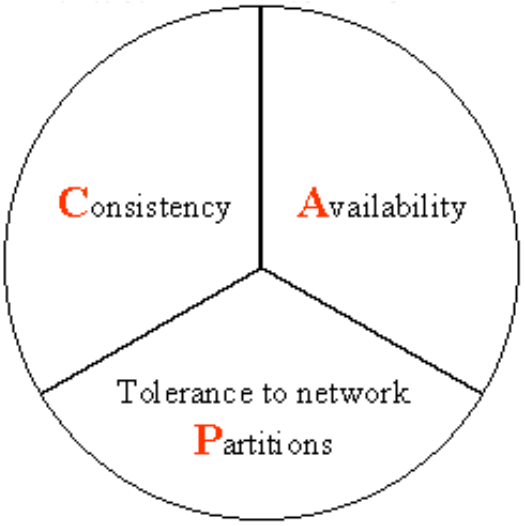
序

日前国内没有一套比较完整的NoSQL数据库资料，有很多先驱整理发表了很多，但不是很系统。不材尝试着将各家的资料整合一下，并书写了一些自己的见解。
本书写了一些目前的NoSql的一些主要技术，算法和思想。同时列举了大量的现有的数据库实例。读完全篇，相信读者会对NoSQL数据库了解个大概。
另外我还准备开发一个开源内存数据库galaxydb.本书也是为这个数据库提供一些架构资料。

思想篇

CAP，BASE和最终一致性是NoSQL数据库存在的三大基石。而五分钟法则是内存数据存储了理论依据。这个是一切的源头。

CAP



- C: Consistency 一致性
- A: Availability 可用性(指的是快速获取数据)
- P: Tolerance of network Partition 分区容忍性(分布式)

10年前，Eric Brewer教授指出了著名的CAP理论，后来Seth Gilbert 和 Nancy lynch两人证明了CAP理论的正确性。CAP理论告诉我们，一个分布式系统不可能满足一致性，可用性和分区容错性这三个需求，最多只能同时满足两个。
熊掌与鱼不可兼得也。关注的是一致性，那么您就需要处理因为系统不可用而导致的写操作失败的情况，而如果您关注的是可用性，那么您应该知道系统的read操作可能不能精确的读取到write操作写入的最新值。因此系统的关注点不同，相应的采用的策略也是不一样的，只有真正的理解了系统的需求，才有可能利用好CAP理论。

作为架构师，一般有两个方向来利用CAP理论

1. key-value存储，如Amaze Dynamo等，可根据CAP三原则灵活选择不同倾向的数据库产品。
2. 领域模型 + 分布式缓存 + 存储 （Qi4j和NoSql运动），可根据CAP三原则结合自己项目定制灵活的分布式方案，难度高。

我准备提供第三种方案：实现可以配置CAP的数据库，动态调配CAP。

- CA：传统关系数据库
- AP：key-value数据库

而对大型网站，可用性与分区容忍性优先级要高于数据一致性，一般会尽量朝着 A、P 的方向设计，然后通过其它手段保证对于一致性的商务需求。架构设计师不要精力浪费在如何设计能满足三者的完美分布式系统，而是应该进行取舍。
不同数据对于一致性的要求是不同的。举例来讲，用户评论对不一致是不敏感的，可以容忍相对较长时间的不一致，这种不一致并不会影响交易和用户体验。而产品价格数据则是非常敏感的，通常不能容忍超过10秒的价格不一致。

CAP理论的证明：Brewer's CAP Theorem

最终一致性

一言以蔽之：过程松，结果紧，最终结果必须保持一致性

为了更好的描述客户端一致性，我们通过以下的场景来进行，这个场景中包括三个组成部分：

- 存储系统

  存储系统可以理解为一个黑盒子，它为我们提供了可用性和持久性的保证。

- Process A

  ProcessA主要实现从存储系统write和read操作

- Process B 和ProcessC

  ProcessB和C是独立于A，并且B和C也相互独立的，它们同时也实现对存储系统的write和read操作。

下面以上面的场景来描述下不同程度的一致性：

- 强一致性

  强一致性（即时一致性） 假如A先写入了一个值到存储系统，存储系统保证后续A,B,C的读取操作都将返回最新值

- 弱一致性

  假如A先写入了一个值到存储系统，存储系统不能保证后续A,B,C的读取操作能读取到最新值。此种情况下有一个"不一致性窗口"的概念，它特指从A写入值，到后续操作A,B,C读取到最新值这一段时间。

- 最终一致性

  最终一致性是弱一致性的一种特例。假如A首先write了一个值到存储系统，存储系统保证如果在A,B,C后续读取之前没有其它写操作更新同样的值的话，最终所有的读取操作都会读取到最A写入的最新值。此种情况下，如果没有失败发生的话，"不一致性窗口"的大小依赖于以下的几个因素：交互延迟，系统的负载，以及复制技术中replica的个数（这个可以理解为master/salve模式中，salve的个数），最终一致性方面最出名的系统可以说是DNS系统，当更新一个域名的IP以后，根据配置策略以及缓存控制策略的不同，最终所有的客户都会看到最新的值。

变体

- Causal consistency（因果一致性）

  如果Process A通知Process B它已经更新了数据，那么Process B的后续读取操作则读取A写入的最新值，而与A没有因果关系的C则可以最终一致性。

- Read-your-writes consistency

  如果Process A写入了最新的值，那么Process A的后续操作都会读取到最新值。但是其它用户可能要过一会才可以看到。

- Session consistency

  此种一致性要求客户端和存储系统交互的整个会话阶段保证Read-your-writes consistency.Hibernate的session提供的一致性保证就属于此种一致性。

- Monotonic read consistency

  此种一致性要求如果Process A已经读取了对象的某个值，那么后续操作将不会读取到更早的值。

- Monotonic write consistency

  此种一致性保证系统会序列化执行一个Process中的所有写操作。

BASE

说起来很有趣，BASE的英文意义是碱，而ACID是酸。真的是水火不容啊。

- Basically Availble --基本可用
- Soft-state --软状态/柔性事务

  "Soft state" 可以理解为"无连接"的, 而 "Hard state" 是"面向连接"的

- Eventual Consistency --最终一致性

  最终一致性， 也是是 ACID 的最终目的。

BASE模型反ACID模型，完全不同ACID模型，牺牲高一致性，获得可用性或可靠性： Basically Available基本可用。支持分区失败(e.g. sharding碎片划分数据库) Soft state软状态 状态可以有一段时间不同步，异步。Eventually consistent最终一致，最终数据是一致的就可以了，而不是时时一致。

BASE思想的主要实现有
1.按功能划分数据库

2.sharding碎片

BASE思想主要强调基本的可用性，如果你需要高可用性，也就是纯粹的高性能，那么就要以一致性或容错性为牺牲，BASE思想的方案在性能上还是有潜力可挖的。

其他

I/O的五分钟法则

在 1987 年，Jim Gray 与 Gianfranco Putzolu 发表了这个"五分钟法则"的观点，简而言之，如果一条记录频繁被访问，就应该放到内存里，否则的话就应该待在硬盘上按需要再访问。这个临界点就是五分钟。 看上去像一条经验性的法则，实际上五分钟的评估标准是根据投入成本判断的，根据当时的硬件发展水准，在内存中保持 1KB 的数据成本相当于硬盘中存据 400 秒的开销(接近五分钟)。这个法则在 1997 年左右的时候进行过一次回顾，证实了五分钟法则依然有效（硬盘、内存实际上没有质的飞跃)，而这次的回顾则是针对 SSD 这个"新的旧硬件"可能带来的影响。

TABLE 3 **Break-even Intervals (seconds)**

Page size	1 KB	4 KB	16 KB	64 KB	256 KB
RAM-SATA	20,978	5,248	1,316	334	88
RAM-flash	2,513	876	467	365	339
Flash-SATA	32,253	8,070	2,024	513	135
RAM-$400	1,006	*351*	187	146	136
$400-SATA	80,553	20,155	5,056	1,281	*337*

随着闪存时代的来临，五分钟法则一分为二：是把 SSD 当成较慢的内存（extended buffer pool ）使用还是当成较快的硬盘（extended disk）使用。小内存页在内存和闪存之间的移动对比大内存页在闪存和磁盘之间的移动。在这个法则首次提出的 20 年之后，在闪存时代，5 分钟法则依然有效，只不过适合更大的内存页(适合 64KB 的页，这个页大小的变化恰恰体现了计算机硬件工艺的发展，以及带宽、延时)。

不要删除数据

Oren Eini（又名Ayende Rahien）建议开发者尽量避免数据库的软删除操作，读者可能因此认为硬删除是合理的选择。作为对Ayende文章的回应，Udi Dahan强烈建议完全避免数据删除。

所谓软删除主张在表中增加一个IsDeleted列以保持数据完整。如果某一行设置了IsDeleted标志列，那么这一行就被认为是已删除的。Ayende觉得这种方法"简单、容易理解、容易实现、容易沟通"，但"往往是错的"。问题在于：

删除一行或一个实体几乎总不是简单的事件。它不仅影响模型中的数据，还会影响模型的外观。所以我们才要有外键去确保不会出现"订单行"没有对应的父"订单"的情况。而这个例子只能算是最简单的情况。.....

当采用软删除的时候，不管我们是否情愿，都很容易出现数据受损，比如谁都不在意的一个小调整，就可能使"客户"的"最新订单"指向一条已经软删除的订单。

如果开发者接到的要求就是从数据库中删除数据，要是不建议用软删除，那就只能硬删除了。为了保证数据一致性，开发者除了删除直接有关的数据行，还应该级联地删除相关数据。可Udi Dahan提醒读者注意，真实的世界并不是级联的：

假设市场部决定从商品目录中删除一样商品，那是不是说所有包含了该商品的旧订单都要一并消失？再级联下去，这

些订单对应的所有发票是不是也该删除？这么一步步删下去，我们公司的损益报表是不是应该重做了？

没天理了。

问题似乎出在对"删除"这词的解读上。Dahan给出了这样的例子：

我说的"删除"其实是指这产品"停售"了。我们以后不再卖这种产品，清掉库存以后不再进货。以后顾客搜索商品或者翻阅目录的时候不会再看见这种商品，但管仓库的人暂时还得继续管理它们。"删除"是个贪方便的说法。

他接着举了一些站在用户角度的正确解读：

订单不是被删除的，是被"取消"的。订单取消得太晚，还会产生花费。

员工不是被删除的，是被"解雇"的（也可能是退休了）。还有相应的补偿金要处理。

职位不是被删除的，是被"填补"的（或者招聘申请被撤回）。

在上面这些例子中，我们的着眼点应该放在用户希望完成的任务上，而非发生在某个
实体身上的技术动作。几乎在所有的情况下，需要考虑的实体总不止一个。

为了代替IsDeleted标志，Dahan建议用一个代表相关数据状态的字段：有效、停用、取消、弃置等等。用户可以借助这样一个状态字段回顾过去的数据，作为决策的依据。

删除数据除了破坏数据一致性，还有其它负面的后果。Dahan建议把所有数据都留在数据库里："别删除。就是别删除。"

RAM是硬盘,硬盘是磁带

Jim Gray在过去40年中对技术发展有过巨大的贡献，"内存是新的硬盘，硬盘是新的磁带"是他的名言。"实时"Web应用不断涌现，达到海量规模的系统越来越多，这种后浪推前浪的发展模式对软硬件又有何影响？

Tim Bray早在网格计算成为热门话题之前，就讨论过以RAM和网络为中心的硬件结构的优势，可以用这种硬件建立比磁盘集群速度更快的RAM集群。

> 对于数据的随机访问，内存的速度比硬盘高几个数量级（即使是最高端的磁盘存储系统也只是勉强达到1,000次寻道/秒）。其次， 随着数据中心的网络速度提高，访问内存的成本更进一步降低。通过网络访问另一台机器的内存比访问磁盘成本更低。就在我写下这段话的时候，Sun的 Infiniband产品线中有一款具备9个全互联非阻塞端口交换机，每个端口的速度可以达到30Gbit/sec！ Voltaire产品的端口甚至更多；简直不敢想象。（如果你想了解这类超高性能网络的最新进展，请关注Andreas Bechtolsheim在Standford开设的课程。）

各种操作的时间，以2001年夏季，典型配置的 1GHz 个人计算机为标准：

执行单一指令	1 纳秒
从L1 高速缓存取一个字	2 纳秒
从内存取一个字	10 纳秒
从磁盘取连续存放的一个字	200 纳秒
磁盘寻址并取字	8 毫秒
以太网	2GB/s

Tim还指出Jim Gray的
名言中后半句所阐述的真理："对于随机访问，硬盘慢得不可忍受；但如果你把硬盘当成磁带来用，它吞吐连续数据的速率令人震惊；它天生适合用来给以RAM为主的应用做日志（logging and journaling）。"

时间闪到几年之后的今天，我们发现硬件的发展趋势在RAM和网络领域势头不减，而在硬盘领域则止步不前。Bill McColl提到用于并行计算的海量内存系统已经出现：

> 内存是新的硬盘！硬盘速度提高缓慢，内存芯片容量指数上升，in-memory软件架构有望给各类数据密集的应用带来数量级的性能提升。小型机架服务器（1U、2U）很快就会具备T字节、甚至更大量的内存，这将会改变服务器架构中内存和硬盘之间的平衡。硬盘将成为新的磁带，像磁带一样作为顺序存储介质使用（硬盘的顺序访问相当快速），而不再是随机存储介质（非常慢）。这里面有着大量的机会，新产品的性能有望提高10倍、100倍。

Dare Obsanjo指出如果不把这句真言当回事，会带来什么样的恶劣后果—— 也就是Twitter正面临的麻烦。论及Twitter的内容管理，Obsanjo说，“如果一个设计只是简单地反映了问题描述，你去实现它就会落入磁盘 I/O的地狱。不管你用Ruby on Rails、Cobol on Cogs、C++还是手写汇编都一样，读写负载照样会害死你。”换言之，应该把随机操作推给RAM，只给硬盘留下顺序操作。

Tom White是Hadoop Core项目的提交者，也是Hadoop项目管理委员会的成员。他对Gray的真言中“硬盘是新的磁带”部分作了更深入地探讨。White在讨论MapReduce编程模型的时候指出，为何对于Hadloop这类工具来说，硬盘仍然是可行的应用程序数据存储介质：

> 本质上，在MapReduce的工作方式中，数据流式地读出和写入硬盘，MapReduce是以硬盘的传输速率不断地对这些数据进行排序和合并。 与之相比，访问关系数据库中的数据，其速率则是硬盘的寻道速率（寻道指移动磁头到盘面上的指定位置读取或写入数据的过程）。为什么要强调这一点？请看看寻道时间和磁盘传输率的发展曲线。寻道时间每年大约提高5%，而数据传输率每年大约提高20%。寻道时间的进步比数据传输率慢——因此采用由数据传输率决定性能的模型是有利的。MapReduce正是如此。

虽然固态硬盘（SSD）能否改变寻道时间/传输率的对比还有待观察，White文章的跟贴中，很多人都认为SSD会成为RAM/硬盘之争中的平衡因素。

Nati Shalom对内存和硬盘在数据库部署和使用中的角色作了一番有理有据的评述。 Shalom着重指出用数据库集群和分区来解决性能和可伸缩性的局限。他说，“数据库复制和数据库分区都存在相同的基本问题，它们都依赖于文件系统/硬盘 的性能，建立数据库集群也非常复杂”。他提议的方案是转向In-Memory Data Grid（IMDG），用Hibernate二级缓存或者GigaSpaces Spring DAO之类的技术作支撑，将持久化作为服务（Persistence as a Service）提供给应用程序。Shalom解释说，IMDG

> 提供在内存中的基于对象的数据库能力，支持核心的数据库功能，诸如高级索引和查询、事务语义和锁。IMDG还从应用程序的代码中抽象出了数据的拓扑。通过这样的方式，数据库不会完全消失，只是挪到了“正确的”位置。

IMDG相比直接RDBMS访问的优势列举如下：

- 位于内存中，速度和并发能力都比文件系统优越得多
- 数据可通过引用访问
- 直接对内存中的对象执行数据操作
- 减少数据的争用
- 并行的聚合查询
- 进程内（In-process）的局部缓存
- 免除了对象-关系映射（ORM）

你是否需要改变对应用和硬件的思维方式，最终取决于你要用它们完成的工作。但似乎公论认为，开发者解决性能和可伸缩性的思路已经到了该变一变的时候。

Amdahl定律和Gustafson定律

这里，我们都以S(n)表示n核系统对具体程序的加速比，K表示串行部分计算时间比例。

Amdahl 定律的加速比：S(n) = 使用1个处理器的串行计算时间 / 使用n个处理器的并行计算时间
S(n) = 1/(K+(1-K)/n) = n/(1+(n-1)K)
Gustafson定律的加速比：S(n) = 使用n个处理器的并行计算量 / 使用1个处理器的串行计算量
S(n) = K+(1-K)n
有点冷是不是？

通俗的讲，Amdahl 定律将工作量看作1，有n核也只能分担1-K的工作量；而Gustafson定律则将单核工作量看作1，有n核，就可以增加n(1-K)的工作量。

这里没有考虑引进分布式带来的开销，比如网络和加锁。成本还是要仔细核算的，不是越分布越好。

控制算法的复杂性在常数范围之内。

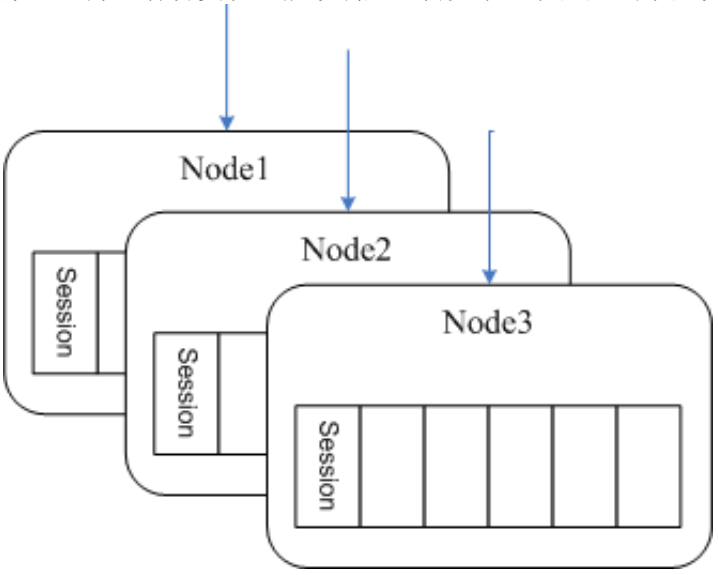
万兆以太网

手段篇

一致性哈希

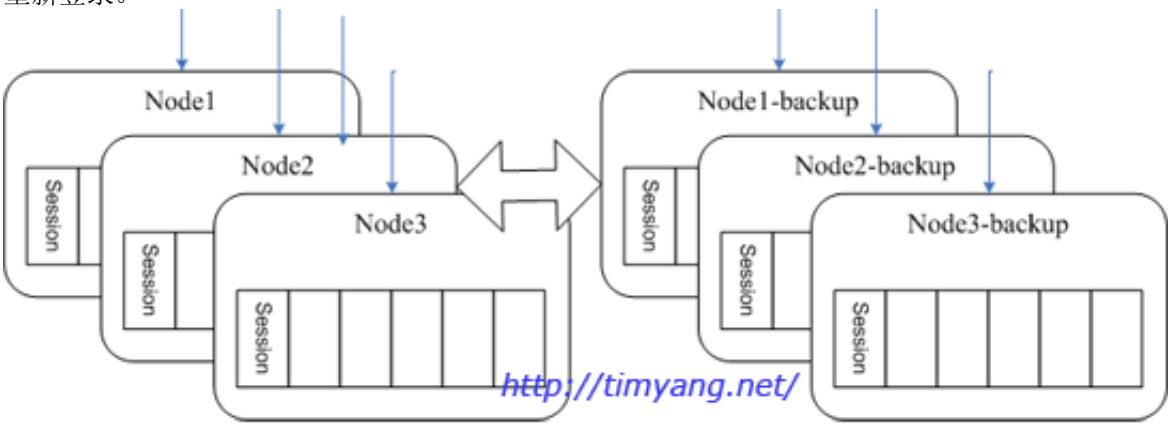
要求分布式架构的发展说起。

第一阶段
考虑到单服务器不能承载，因此使用了分布式架构，最初的算法为 hash() mod n, hash()通常取用户ID，n为节点数。此方法容易实现且能够满足运营要求。缺点是当单点发生故障时，系统无法自动恢复。



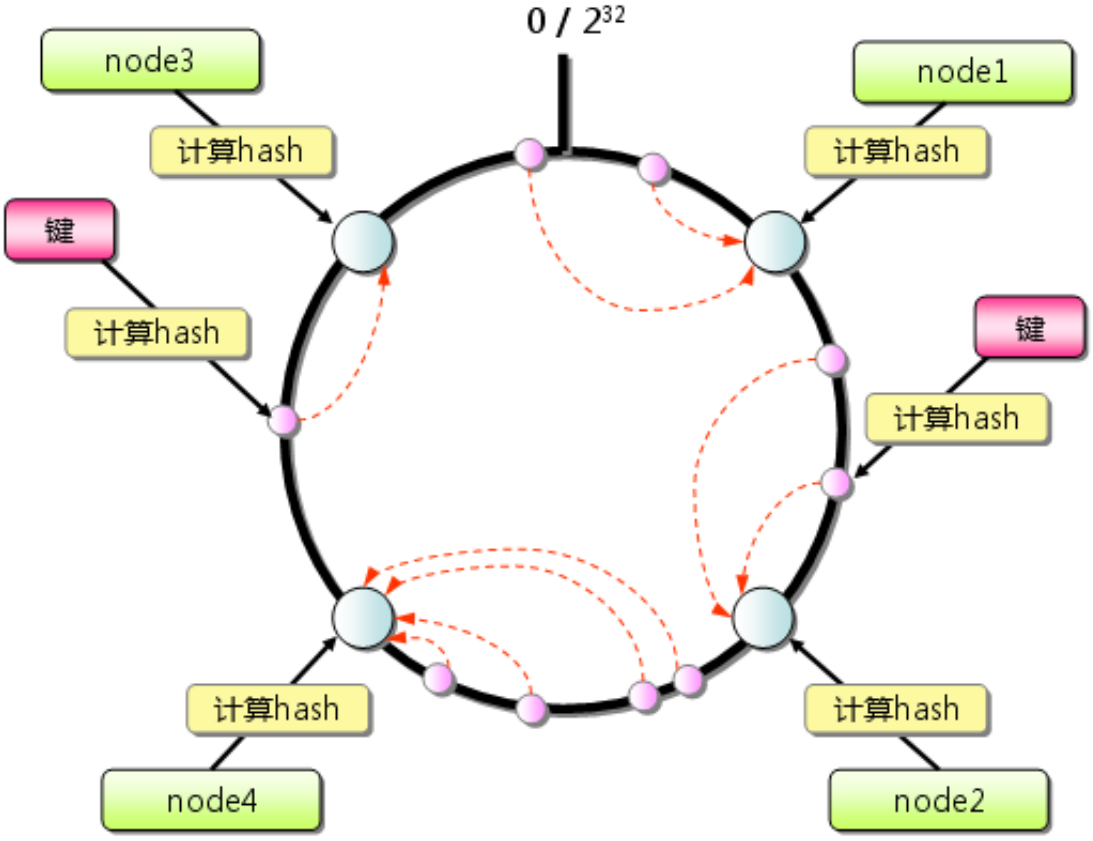
第二阶段
为了解决单点故障，使用 hash() mod (n/2), 这样任意一个用户都有2个服务器备选，可由client随机选取。由于不同服务器之间的用户需要彼此交互，所以所有的服务器需要确切的知道用户所在的位置。因此用户位置被保存到memcached中。

当一台发生故障，client可以自动切换到对应backup，由于切换前另外1台没有用户的session，因此需要client自行重新登录。



这个阶段的设计存在以下问题
负载不均衡，尤其是单台发生故障后剩下一台会压力过大。
不能动态增删节点
节点发生故障时需要client重新登录

第三阶段
打算去掉硬编码的hash() mod n 算法，改用一致性哈希(consistent hashing)分布
假如采用Dynamo中的strategy 1
我们把每台server分成v个虚拟节点，再把所有虚拟节点(n*v)随机分配到一致性哈希的圆环上，这样所有的用户从自己圆环上的位置顺时针往下取到第一个vnode就是自己所属节点。当此节点存在故障时，再顺时针取下一个作为替代节点。



优点：发生单点故障时负载会均衡分散到其他所有节点，程序实现也比较优雅。

亚马逊的现状

aw2.0公司的Alan Williamson撰写了一篇报道，主要是关于他在Amazon EC2上的体验的，他抱怨说，Amazon是公司唯一使用的云提供商，看起来它在开始时能够适应得很好，但是有一个临界点：

> 在开始的日子里Amazon的表现非常棒。实例在几分钟内启动，几乎没有遇到任何问题，即便是他们的小实例（SMALL INSTANCE）也很健壮，足以支持适当使用的MySQL数据库。在20个月内，Amazon云系统一切运转良好，不需要任何的关心和抱怨。
> ……
> 然而，在最后的八个月左右，他们"盔甲"内的漏洞开始呈现出来了。第一个弱点前兆是，新加入的Amazon SMALL实例的性能出现了问题。根据我们的监控，在服务器场中新添加的机器，与原先的那些相比性能有所下降。开始我们认为这是自然出现的怪现象，只是碰 巧发生在"吵闹的邻居"（Noisy Neighbors）旁边。根据随机法则，一次快速的停机和重新启动经常就会让我们回到"安静的邻居"旁边，那样我们可以达到目的。
> …… 然而，在最后的一两个月中，我们发现，甚至是这些"使用高级CPU的中等实例"也遭受了与小

实例相同的命运，其中，新的实例不管处于什么位置，看起来似乎都表现得一样。经过调查，我们还发现了一个新问题，它已经悄悄渗透到到Amazon的世界中，那就是内部网络延迟。

算法的选择

不同的哈希算法可以导致数据分布的不同位置，如果十分均匀，那么一次MapReduce就涉及节点较多，但热点均匀，方便管理。反之，热点不均，会大致机器效率发挥不完全。

Quorum NRW

Write to all copies (with latest version no), wait synchronously for W success
Read from all copies, wait for first R responses, pick the highest version number

Condition: W + R > N

- N: 复制的节点数量
- R: 成功读操作的最小节点数
- W: 成功写操作的最小节点数

只需W + R > N，就可以保证强一致性。

第一个关键参数是 N，这个 N 指的是数据对象将被复制到 N 台主机上，N 在实例级别配置，协调器将负责把数据复制到 N-1 个节点上。N 的典型值设置为 3.

复 制中的一致性，采用类似于 Quorum 系统的一致性协议实现。这个协议有两个关键值：R 与 W。R 代表一次成功的读取操作中最小参与节点数量，W 代表一次成功的写操作中最小参与节点数量。R + W>N ，则会产生类似 quorum 的效果。该模型中的读(写)延迟由最慢的 R(W)复制决定，为得到比较小的延迟，R 和 W 有的时候的和又设置比 N 小。

如果N中的1台发生故障，Dynamo立即写入到preference list中下一台，确保永远可写入

如 果W+R>N，那么分布式系统就会提供强一致性的保证，因为读取数据的节点和被同步写入的节点是有重叠的。在一个RDBMS的复制模型中 （Master/salve），假如N=2,那么W=2,R=1此时是一种强一致性,但是这样造成的问题就是可用性的减低，因为要想写操作成功，必须要等 2个节点都完成以后才可以。

在分布式系统中，一般都要有容错性，因此一般N都是大于3的，此时根据CAP理论，一致性，可用性和分区容错 性最多只能满足两个，那么我们就需要在一致性和分区容错性之间做一平衡，如果要高的一致性，那么就配置N=W，R=1,这个时候可用性就会大大降低。如果 想要高的可用性，那么此时就需要放松一致性的要求，此时可以配置W=1，这样使得写操作延迟最低，同时通过异步的机制更新剩余的N-W个节点。

当存储系统保证最终一致性时，存储系统的配置一般是W+R<=N,此时读取和写入操作是不重叠的，不一致性的窗口就依赖于存储系统的异步实现方式，不一致性的窗口大小也就等于从更新开始到所有的节点都异步更新完成之间的时间。

(N,R,W) 的值典型设置为 (3, 2 ,2),兼顾性能与可用性。R 和 W 直接影响性能、扩展性、一致性，如果 W 设置 为 1，则一个实例中只要有一个节点可用，也不会影响写操作，如果 R 设置为 1 ，只要有一个节点可用，也不会影响读请求，R 和 W 值过小则影响一致性，过大也不好，这两个值要平衡。对于这套系统的典型的 SLA 要求 99.9% 的读写操作在 300ms 内完成。

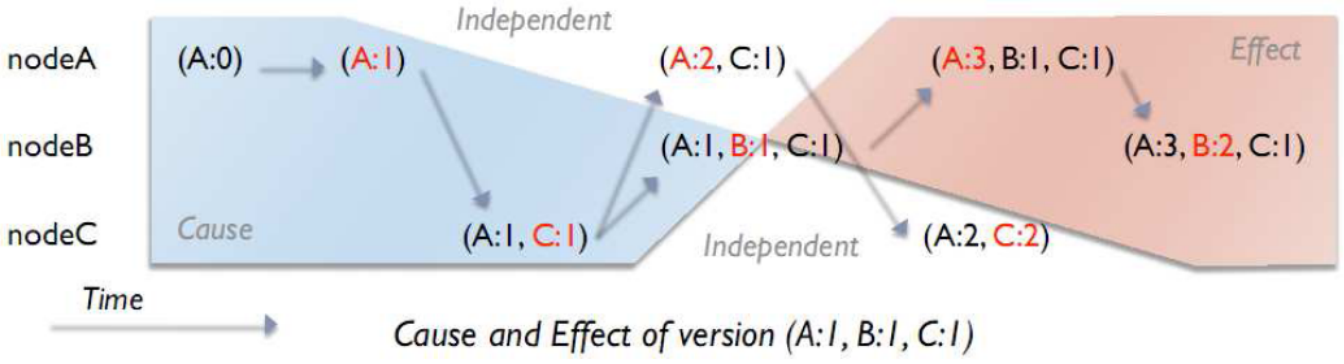
无 论是Read-your-writes-consistency,Session consistency,Monotonic read consistency,它们都通过黏贴（stickiness)客户端到执行分布式请求的服务器端来实现的，这种方式简单是简单，但是它使得负载均衡以 及分区容错变的更加难于管理，有时候也可以通过客户端来实现Read-your-writes-consistency和Monotonic read consistency,此时需要对写的操作的数据加版本号，这样客户端就可以遗弃版本号小于最近看到的版本号的数据。

在系统开发过程 中，根据CAP理论，可用性和一致性在一个大型分区容错的系统中只能满足一个，因此为了高可用性，我们必须放低一致性的要求，但是不同的系统保证的一致性 还是有差别的，这就要求开发者要清楚自己用的系统提供什么样子的最终一致性的保证，一个非常流行的例子就是web应用系统，在大多数的web应用系统中都 有"用户可感知一致性"的概念，这也就是说最终一致性中的"一致性窗口"大小要小于用户下一次的请求，在下次读取操作来之前，数据可以在存储的各个节点之 间复制。还比如假如存储系统提供了

read-your-write-consistency一致性，那么当一个用户写操作完成以后可以立马看到自己的更 新，但是其它的用户要过一会才可以看到更新。

几种特殊情况：
W = 1, R = N,对写操作要求高性能高可用。
R = 1, W = N , 对读操作要求高性能高可用，比如类似cache之类业务。
W = Q, R = Q where Q = N / 2 + 1 一般应用适用，读写性能之间取得平衡。如N=3,W=2,R=2

Vector clock



vector clock算法。可以把这个vector clock想象成每个节点都记录自己的版本信息，而一个数据，包含所有这些版本信息。来看一个例子：假设一个写请求，第一次被节点A处理了。节点A会增加一个版本信息(A，1)。我们把这个时候的数据记做D1(A，1)。 然后另外一个对同样key(这一段讨论都是针对同样的key的)的请求还是被A处理了于是有D2(A，2)。

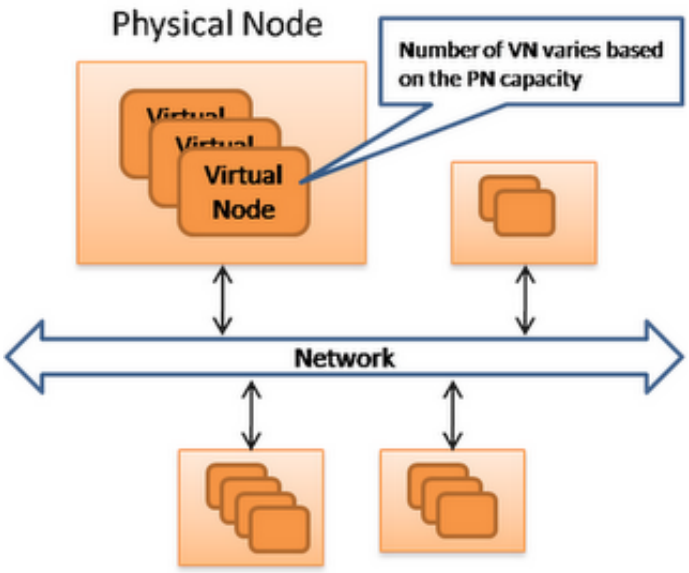
这个时候，D2是可以覆盖D1的，不会有冲突产生。现在我们假设D2传播到了所有节点(B和C)，B和C收到的数据不是从客户产生的，而是别人复制给他们的，所以他们不产生新的版本信息，所以现在B和C都持有数据D2(A，2)。好，继续，又一个请求，被B处理了，生成数据D3(A，2;B，1)，因为这是一个新版本的数据，被B处理，所以要增加B的版本信息。

假设D3没有传播到C的时候又一个请求被C处理记做D4(A，2;C，1)。假设在这些版本没有传播开来以前，有一个读取操作，我们要记得，我们的W=1 那么R=N=3，所以R会从所有三个节点上读，在这个例子中将读到三个版本。A上的D2(A，2);B上的D3(A，2;B，1);C上的D4(A，2;C，1)这个时候可以判断出，D2已经是旧版本，可以舍弃，但是D3和D4都是新版本，需要应用自己去合并。

如果需要高可写性，就要处理这种合并问题。好假设应用完成了冲入解决，这里就是合并D3和D4版本，然后重新做了写入，假设是B处理这个请求，于是有D5(A，2;B，2;C，1);这个版本将可以覆盖掉D1-D4那四个版本。这个例子只举了一个客户的请求在被不同节点处理时候的情况， 而且每次写更新都是可接受的，大家可以自己更深入的演算一下几个并发客户的情况，以及用一个旧版本做更新的情况。

上面问题看似好像可以通过在三个节点里选择一个主节点来解决，所有的读取和写入都从主节点来进行。但是这样就违背了W=1这个约定，实际上还是退化到W=N的情况了。所以如果系统不需要很大的弹性，W=N为所有应用都接受，那么系统的设计上可以得到很大的简化。Dynamo 为了给出充分的弹性而被设计成完全的对等集群(peer to peer)，网络中的任何一个节点都不是特殊的。

Virtual node



虚拟节点，未完成

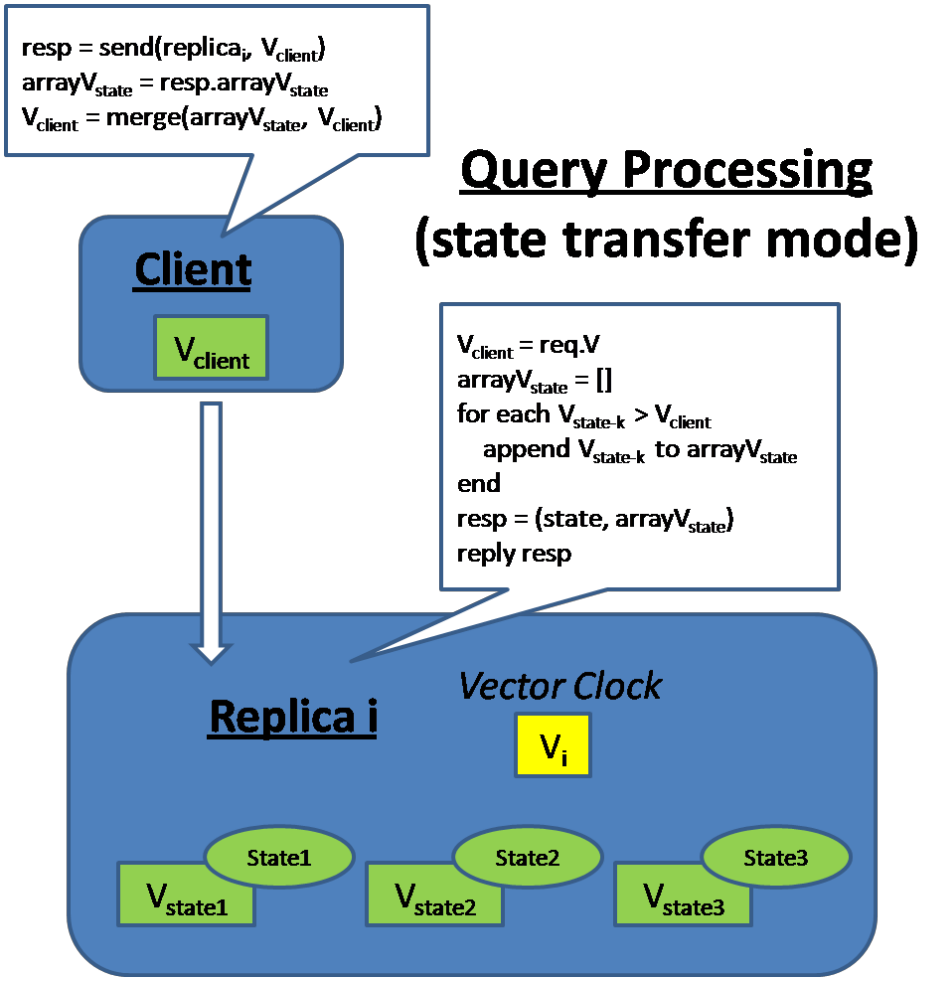
gossip

Gossip协议是一个Gossip思想的P2P实现。现代的分布式系统经常使用这个协议，他往往是唯一的手段。因为底层的结构非常复杂，而且Gossip也很有效。
Gossip协议也被戏称为病毒式传播，因为他的行为生物界的病毒很相似。

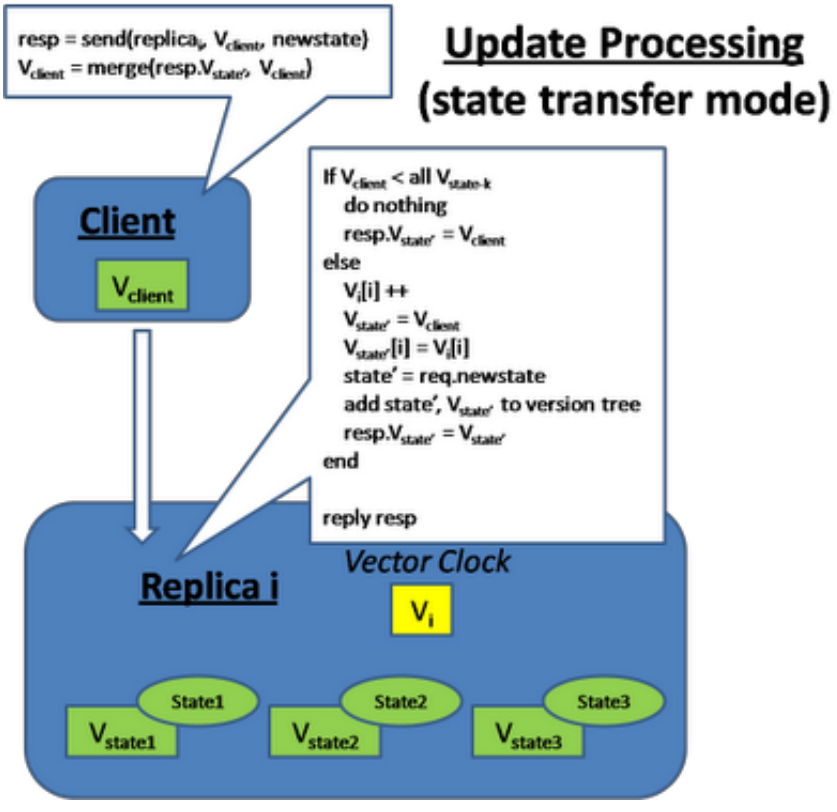
Gossip (State Transfer Model)

在状态转移到模式下，每个重复节点都保持的一个Vector clock和一个state version tree。每个节点的状态都是相同的(based on vector clock comparison),换句话说，state version tree包含有全部的冲突updates.

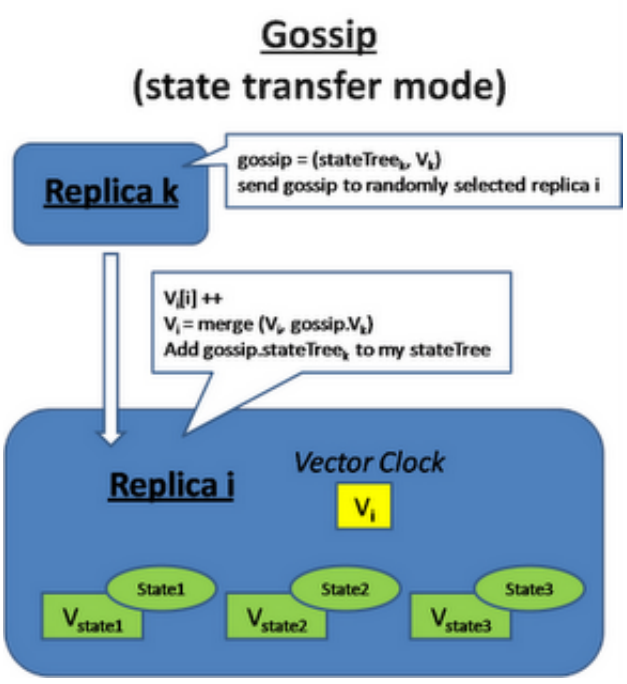
At query time, the client will attach its vector clock and the replica will send back a subset of the state tree which precedes the client's vector clock (this will provide monotonic read consistency). The client will then advance its vector clock by merging all the versions. This means the client is responsible to resolve the conflict of all these versions because when the client sends the update later, its vector clock will precede all these versions.



At update, the client will send its vector clock and the replica will check whether the client state precedes any of its existing version, if so, it will throw away the client's update.



Replicas also gossip among each other in the background and try to merge their version tree together.



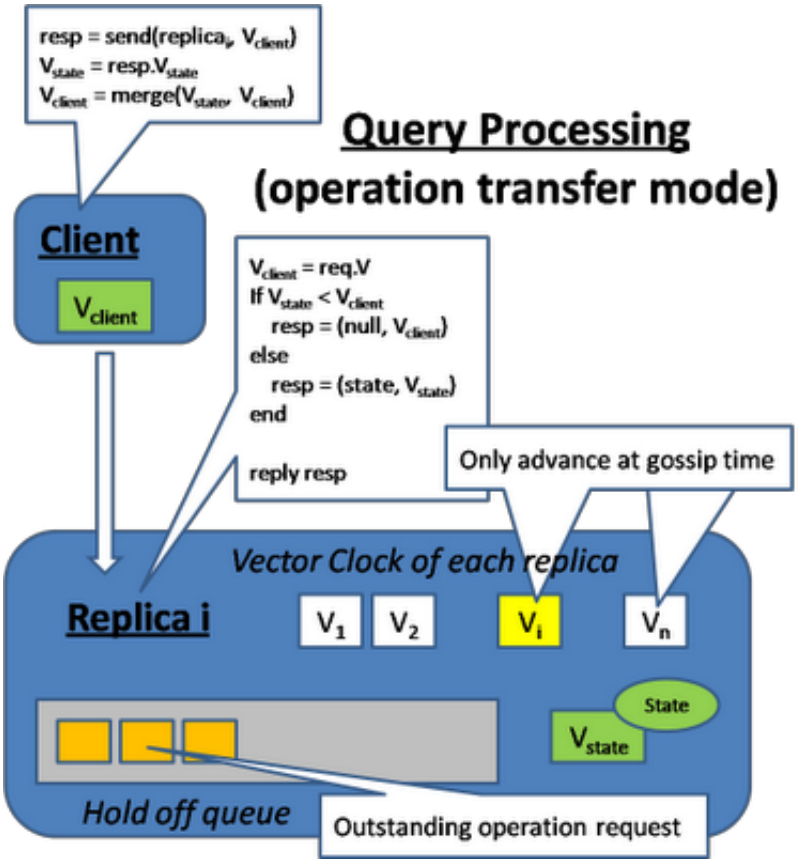
Gossip (Operation Transfer Model)

In an operation transfer approach, the sequence of applying the operations is very important. At the minimum causal order need to be maintained. Because of the ordering issue, each replica has to defer executing the operation until all the preceding operations has been executed. Therefore replicas save the operation request to a log file and exchange the log among each other and consolidate these operation logs to figure out the right sequence to apply the operations to their local store in an appropriate order.

"Causal order" means every replica will apply changes to the "causes" before apply changes to the "effect". "Total order" requires that every replica applies the operation in the same sequence.

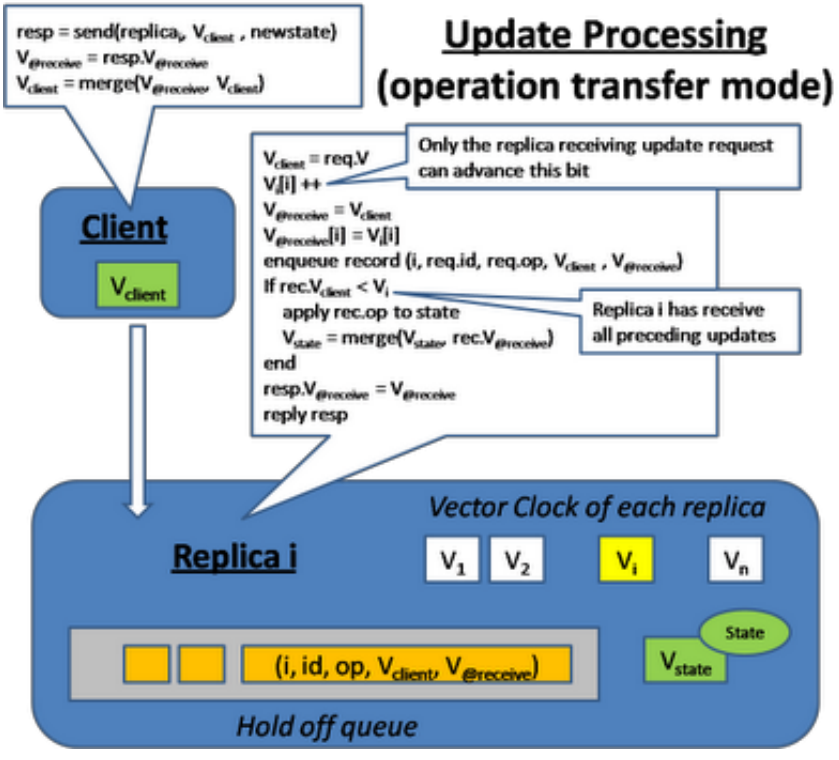
In this model, each replica keeps a list of vector clock, Vi is the vector clock the replica itself and Vj is the vector clock when replica i receive replica j's gossip message. There is also a V-state that represent the vector clock of the last updated state.

When a query is submitted by the client, it will also send along its vector clock which reflect the client's view of the world. The replica will check if it has a view of the state that is later than the client's view.



When an update operation is received, the replica will buffer the update operation until it can be applied to the local state. Every submitted operation will be tag with 2 timestamp, V-client indicates the client's view when he is making the update request. V-@receive is the replica's view when it receives the submission.

This update operation request will be sitting in the queue until the replica has received all the other updates that this one depends on. This condition is reflected in the vector clock Vi when it is larger than V-client



On the background, different replicas exchange their log for the queued updates and update each

other's vector clock. After the log exchange, each replica will check whether certain operation can be applied (when all the dependent operation has been received) and apply them accordingly. Notice that it is possible that multiple operations are ready for applying at the same time, the replica will sort these operation in causal order (by using the Vector clock comparison) and apply them in the right order.

The concurrent update problem at different replica can also happen. Which means there can be multiple valid sequences of operation. In order for different replica to apply concurrent update in the same order, we need a total ordering mechanism.

One approach is whoever do the update first acquire a monotonic sequence number and late comers follow the sequence. On the other hand, if the operation itself is commutative, then the order to apply the operations doesn't matter

After applying the update, the update operation cannot be immediately removed from the queue because the update may not be fully exchange to every replica yet. We continuously check the Vector clock of each replicas after log exchange and after we confirm than everyone has receive this update, then we'll remove it from the queue.

Merkle tree

有数据存储成树状结构，每个节点的Hash是其所有子节点的Hash的Hash，叶子节点的Hash是其内容的Hash。这样一旦某个节点发生变化，其Hash的变化会迅速传播到根节点。需要同步的系统只需要不断查询跟节点的hash，一旦有变化，顺着树状结构就能够在logN级别的时间找到发生变化的内容，马上同步。

Paxos

paxos是一种处理一致性的手段，可以理解为事务吧。
其他的手段不要Google GFS使用的Chubby的Lock service。我不大喜欢那种重型的设计就不费笔墨了。

背景

当规模越来越大的时候。

一、Master/slave

这个是多机房数据访问最常用的方案，一般的需求用此方案即可。因此大家也经常提到“premature optimization is the root of all evil”。
优点：利用mysql replication即可实现，成熟稳定。
缺点：写操作存在单点故障，master坏掉之后slave不能写。另外slave的延迟也是个困扰人的小问题。

二、Multi-master

Multi-master指一个系统存在多个master, 每个master都具有read-write能力，需根据时间戳或业务逻辑合并版本。比如分布式版本管理系统git可以理解成multi-master模式。具备最终一致性。多版本数据修改可以借鉴Dynamo的vector clock等方法。

优点：解决了单点故障。
缺点：不易实现一致性，合并版本的逻辑复杂。

三、Two-phase commit(2PC)

Two-phase commit是一个比较简单的一致性算法。由于一致性算法通常用神话(如Paxos的The Part-Time Parliament论文)来比喻容易理解，下面也举个类似神话的例子。

某班要组织一个同学聚会，前提条件是所有参与者同意则活动举行，任意一人拒绝则活动取消。用2PC算法来执行过程如下

Phase 1

Prepare: 组织者(coordinator)打电话给所有参与者(participant) ，同时告知参与者列表。
Proposal: 提出周六2pm-5pm举办活动。
Vote: participant需vote结果给coordinator：accept or reject。
Block: 如果accept, participant锁住周六2pm-5pm的时间，不再接受其他请求。
Phase 2

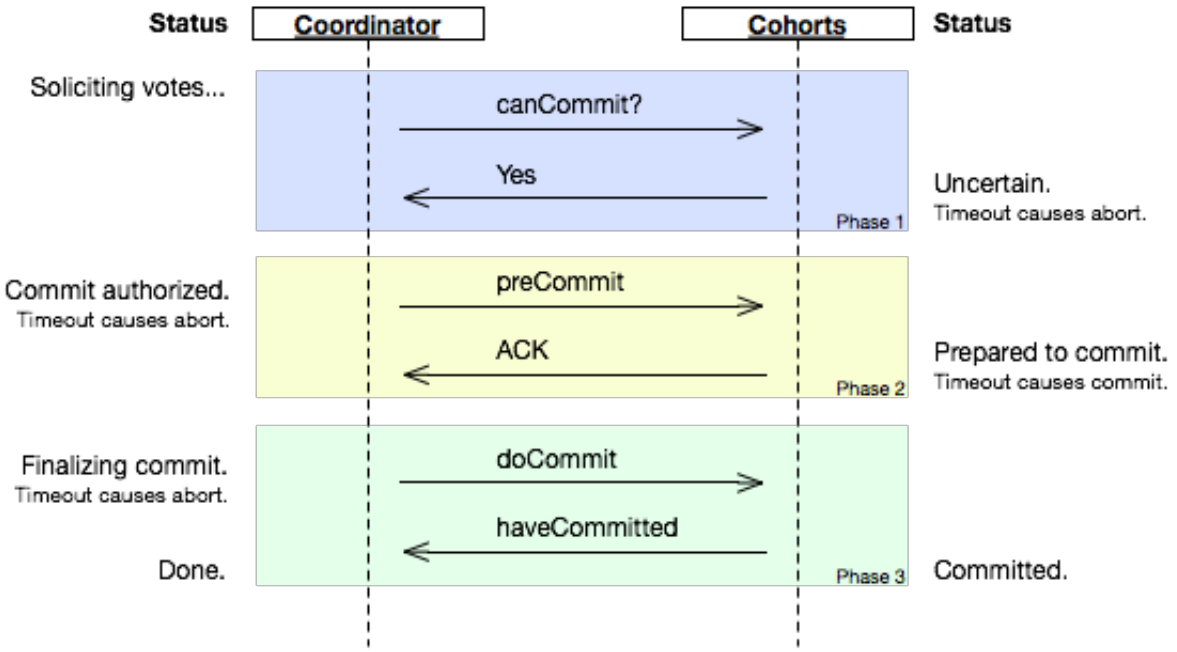
Commit: 如果所有参与者都同意，组织者coodinator通知所有参与者commit, 否则通知abort，participant解除锁定。
Failure 典型失败情况分析

Participant failure:
任一参与者无响应，coordinator直接执行abort
Coordinator failure:
Takeover: 如果participant一段时间没收到cooridnator确认(commit/abort)，则认为coordinator不在了。这时候可自动成为Coordinator备份(watchdog)
Query: watchdog根据phase 1接收的participant列表发起query
Vote: 所有participant回复vote结果给watchdog, accept or reject
Commit: 如果所有都同意，则commit, 否则abort。

优点：实现简单。
缺点：所有参与者需要阻塞(block)，throughput低；无容错机制，一节点失败则整个事务失败。

四、Three-phase commit (3PC)

Three-phase commit是一个2PC的改进版。2PC有一些很明显的缺点，比如在coordinator做出commit决策并开始发送commit之后，某个participant突然crash，这时候没法abort transaction, 这时候集群内实际上就存在不一致的情况，crash恢复后的节点跟其他节点数据是不同的。因此3PC将2PC的commit的过程1分为2,分成preCommit及commit, 如图。



(图片来源：http://en.wikipedia.org/wiki/File:Three-phase_commit_diagram.png)

从图来看，cohorts(participant)收到preCommit之后，如果没收到commit，默认也执行commit，即图上的timeout cause commit。

如果coodinator发送了一半preCommit crash，watchdog接管之后通过query，如果有任一节点收到commit，或者全部节点收到preCommit，则可继续commit，否则abort。

优点：允许发生单点故障后继续达成一致。
缺点：网络分离问题，比如preCommit消息发送后突然两个机房断开，这时候coodinator所在机房会abort，另外剩余replicas机房会commit。

Google Chubby的作者Mike Burrows说过，“there is only one consensus protocol, and that’s Paxos” – all other approaches are just broken versions of Paxos. 意即“世上只有一种一致性算法，那就是Paxos”，所有其他一致性算法都是Paxos算法的不完整版。相比2PC/3PC，Paxos算法的改进
P1a. 每次Paxos实例执行都分配一个编号，编号需要递增，每个replica不接受比当前最大编号小的提案
P2. 一旦一个 value v 被replica通过，那么之后任何再批准的 value 必须是 v，即没有拜占庭将军(Byzantine)问题。拿上面请客的比喻来说，就是一个参与者一旦accept周六2pm-5pm的proposal，就不能改变主意。以后不管谁来问都是accept这个value。
一个proposal只需要多数派同意即可通过。因此比2PC/3PC更灵活，在一个2f+1个节点的集群中，允许有f个节点不可用。

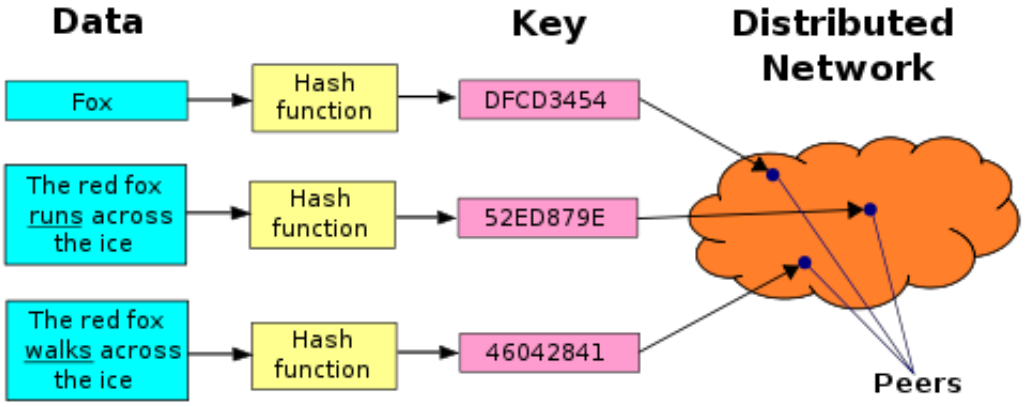
另外Paxos还有很多约束的细节，特别是Google的chubby从工程实现的角度将Paxos的细节补充得非常完整。比如如何避免Byzantine问题，由于节点的持久存储可能会发生故障，Byzantine问题会导致Paxos算法P2约束失效。

以上几种方式原理比较如下

	Backups	M/S	MM	2PC	Paxos
Consistency	Weak	Eventual		Strong	
Transactions	No	Full	Local	Full	
Latency	Low			High	
Throughput	High			Low	Medium
Data loss	Lots	Some		None	
Failover	Down	Read only	Read/write		

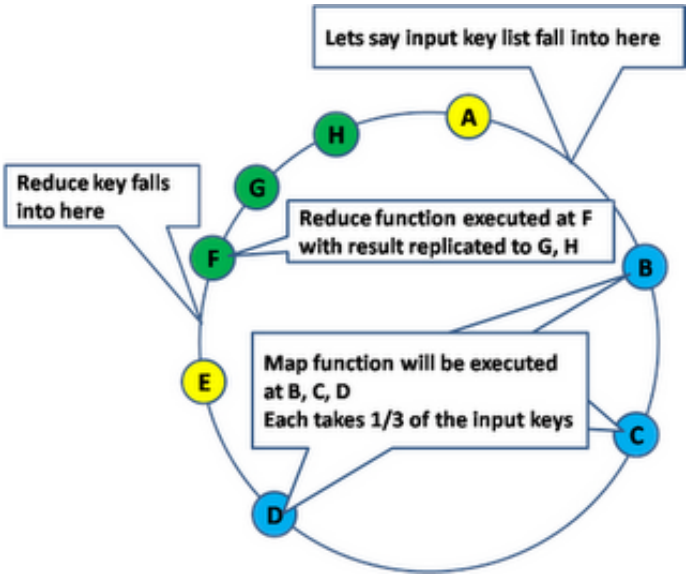
DHT

Distributed hash table



Map Reduce Execution

Map Reduce已经烂大街了，不过还是要提一下。
参见：http://zh.wikipedia.org/wiki/MapReduce



Handling Deletes

但我们执行删除操作的时候必须非常谨慎，以防丢失掉相应的版本信息。

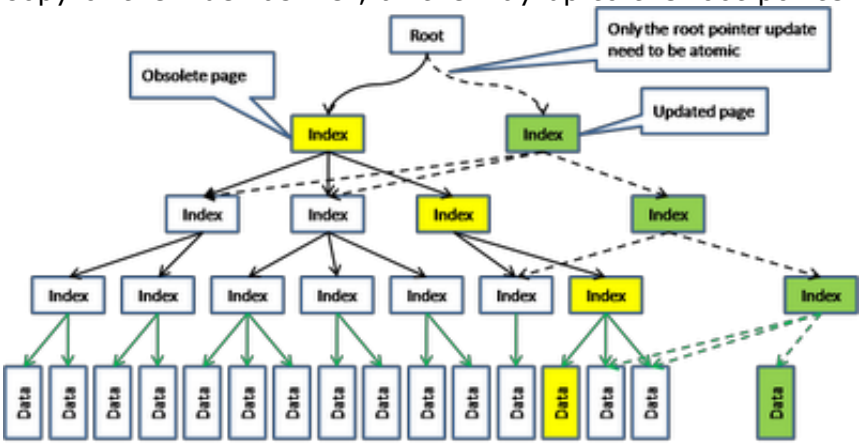
通常我们给一个Object标注上"已删除"的标签。在足够的时间之后，我们在确保版本一致的情况下可以将它彻底删除。回收他的空间。

存储实现

One strategy is to use make the storage implementation pluggable. e.g. A local MySQL DB, Berkeley DB, Filesystem or even a in memory Hashtable can be used as a storage mechanism.

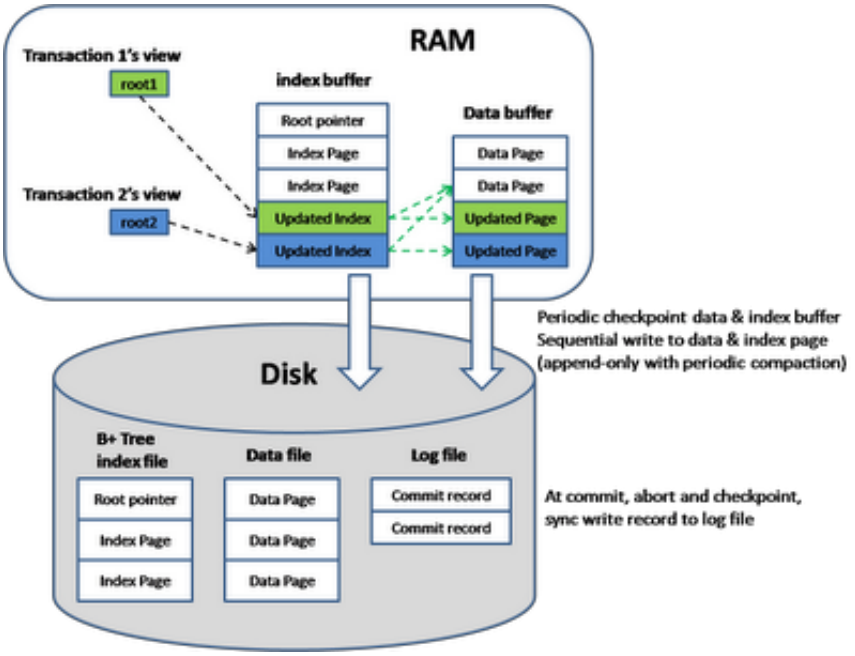
Another strategy is to implement the storage in a highly scalable way. Here are some techniques that I learn from CouchDB and Google BigTable.

CouchDB has a MVCC model that uses a copy-on-modified approach. Any update will cause a private copy being made which in turn cause the index also need to be modified and causing the a private copy of the index as well, all the way up to the root pointer.

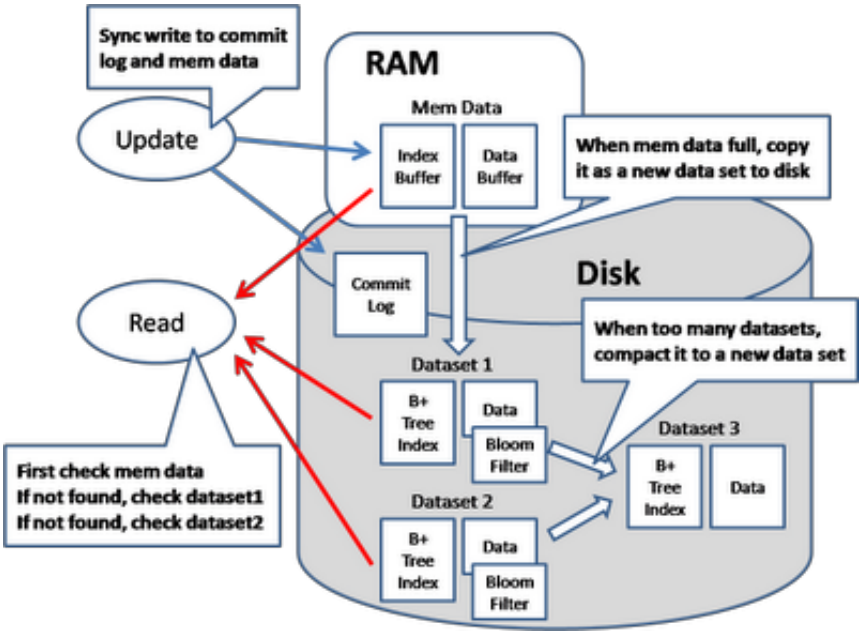


Copy on modify. Everyone sees his own copy of update
Finally the root pointer is swapped and everyone's view is updated
Yellow page becomes garbage over time.
File will be compacted periodically by copying to a different file.

Notice that the update happens in an append-only mode where the modified data is appended to the file and the old data becomes garbage. Periodic garbage collection is done to compact the data. Here is how the model is implemented in memory and disks



In Google BigTable model, the data is broken down into multiple generations and the memory is use to hold the newest generation. Any query will search the mem data as well as all the data sets on disks and merge all the return results. Fast detection of whether a generation contains a key can be done by checking a bloom filter.



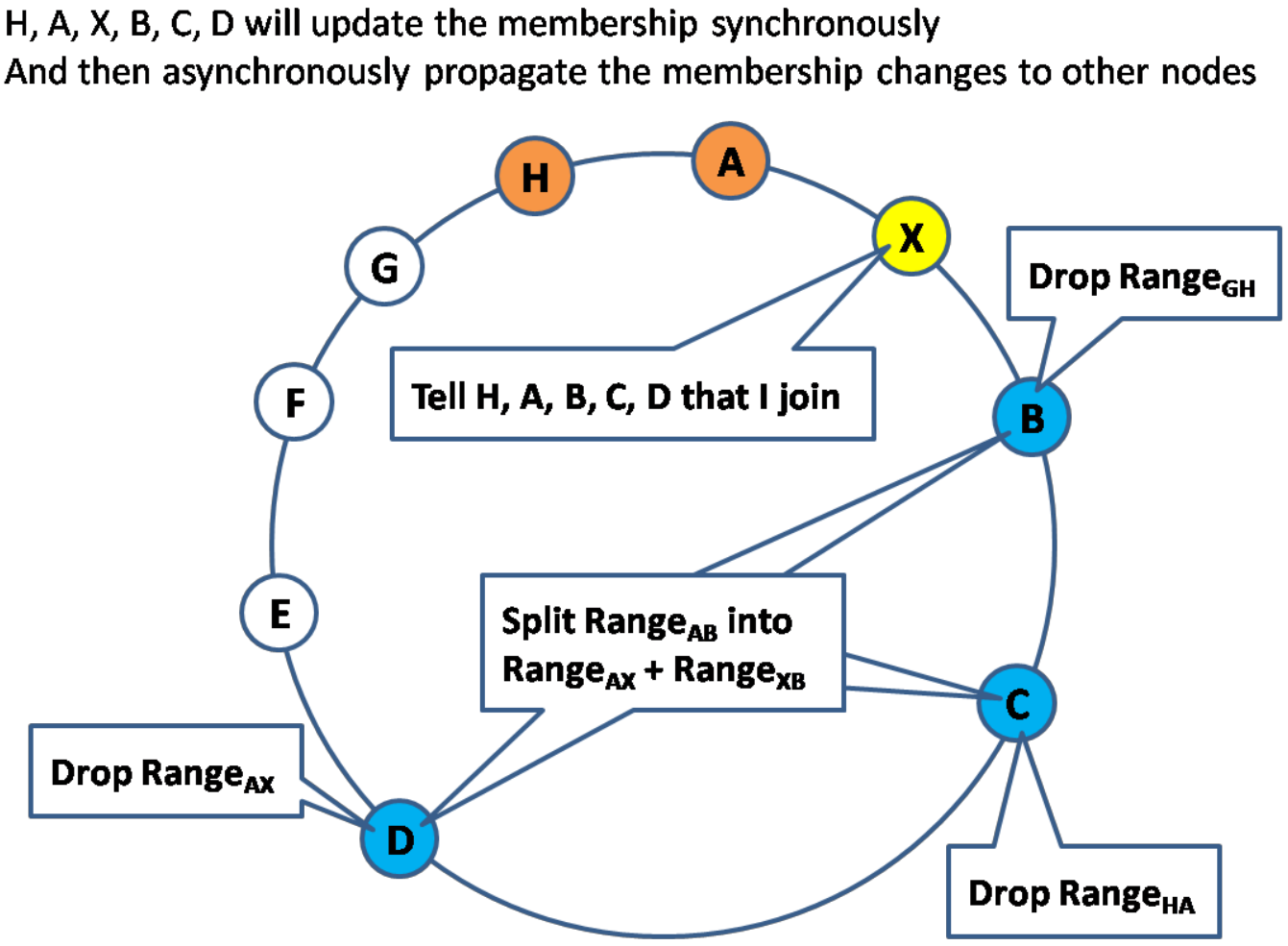
When update happens, both the mem data and the commit log will be written so that if the

节点变化

Notice that virtual nodes can join and leave the network at any time without impacting the operation of the ring.

When a new node joins the network

1. 新加入的节点宣告自己的存在(广播或者其他手段)
2. 他的邻居节点要调整Key的分配和复制关系。这个操作通常是同步的
3. 这个新加入的节点异步的拷贝数据
4. 这个节点变化的操作被发布到其他节点



Notice that other nodes may not have their membership view updated yet so they may still forward the request to the old nodes. But since these old nodes (which is the neighbor of the new joined node) has been updated (in step 2), so they will forward the request to the new joined node.

On the other hand, the new joined node may still in the process of downloading the data and not ready to serve yet. We use the vector clock (described below) to determine whether the new joined node is ready to serve the request and if not, the client can contact another replica.

When an existing node leaves the network (e.g. crash)

1. The crashed node no longer respond to gossip message so its neighbors knows about it.崩溃的节点不再发送Gossip Message的回应，所以他的邻居都知道他是了
2. The neighbor will update the membership changes and copy data asynchronously，他的邻居处理后事，将他的活分给别人干，同时调整节点关系。

We haven't talked about how the virtual nodes is mapped into the physical nodes. Many schemes are possible with the main goal that Virtual Node replicas should not be sitting on the same physical node. One simple scheme is to assigned Virtual node to Physical node in a random manner but check to make sure that a physical node doesn't contain replicas of the same key ranges.

Notice that since machine crashes happen at the physical node level, which has many virtual nodes runs on it. So when a single Physical node crashes, the workload (of its multiple virtual node) is scattered across many physical machines. Therefore the increased workload due to physical node crashes is evenly balanced.

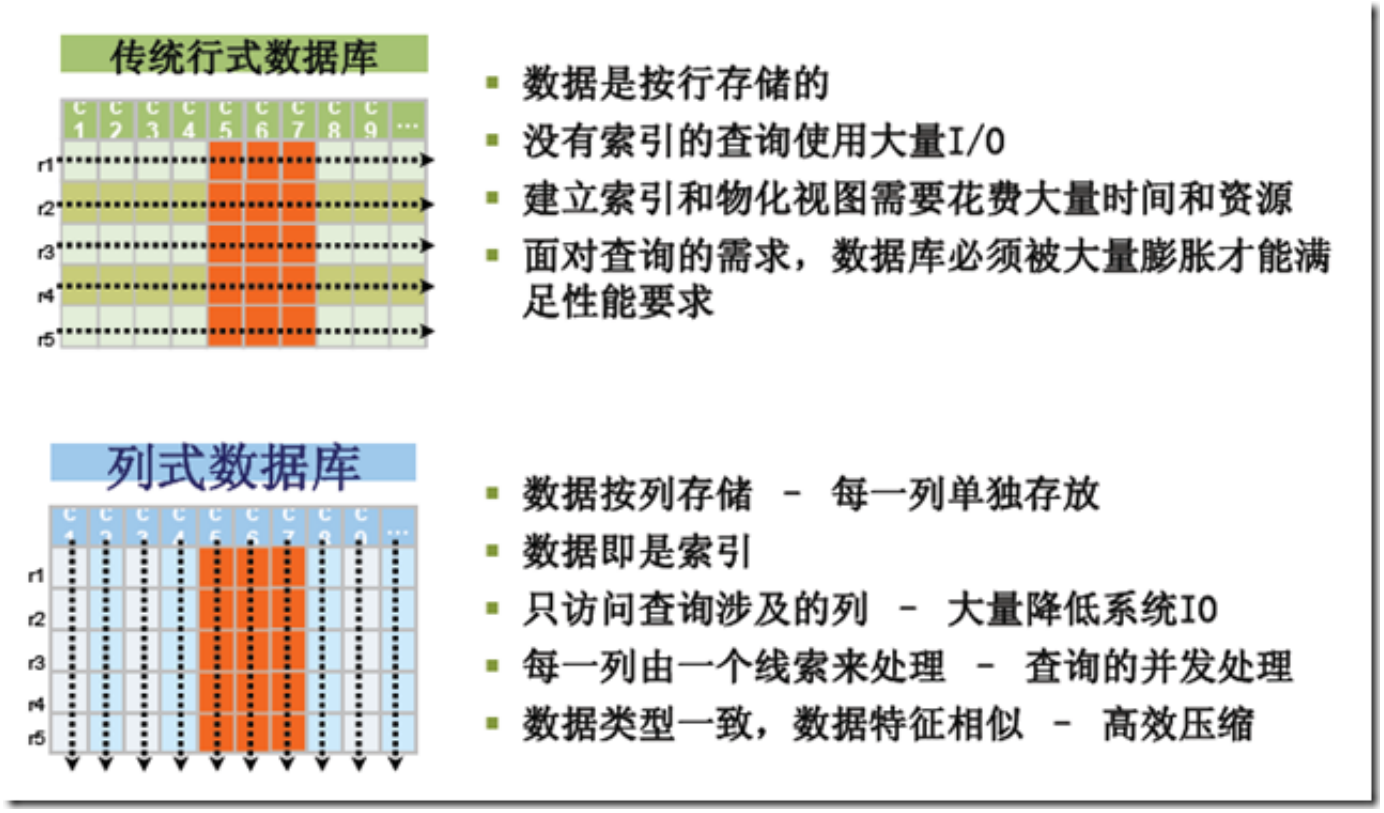
列存

描述

数据库以行、列的二维表的形式存储数据，但是却以一维字符串的方式存储，例如以下的一个表：

EmpId	Lastname	Firstname	Salary
1	Smith	Joe	40000
2	Jones	Mary	50000
3	Johnson	Cathy	44000

这个简单的表包括员工代码(EmpId), 姓名字段(Lastname and Firstname)及工资(Salary).
这个表存储在电脑的内存(RAM)和存储(硬盘)中。虽然内存和硬盘在机制上不同，电脑的操作系统是以同样的方式存储的。数据库必须把这个二维表存储在一系列一维的"字节"中，又操作系统写到内存或硬盘中。
行式数据库把一行中的数据值串在一起存储起来，然后再存储下一行的数据，以此类推。
`1,Smith,Joe,40000;2,Jones,Mary,50000;3,Johnson,Cathy,44000;`
列式数据库把一列中的数据值串在一起存储起来，然后再存储下一列的数据，以此类推。
`1,2,3;Smith,Jones,Johnson;Joe,Mary,Cathy;40000,50000,44000;`



特点

- 良好的压缩比。由于大多数数据库设计都有冗余，如此一来，压缩比非常高，把40多M的数据导入infobright，没想到数据文件只有1M多
- 列上的计算非常的快。
- 方便MapReduce和Key-value模型的融合
- 读取整行的数据较慢，但部分数据较快

简单分析含源码

软件篇

亚数据库

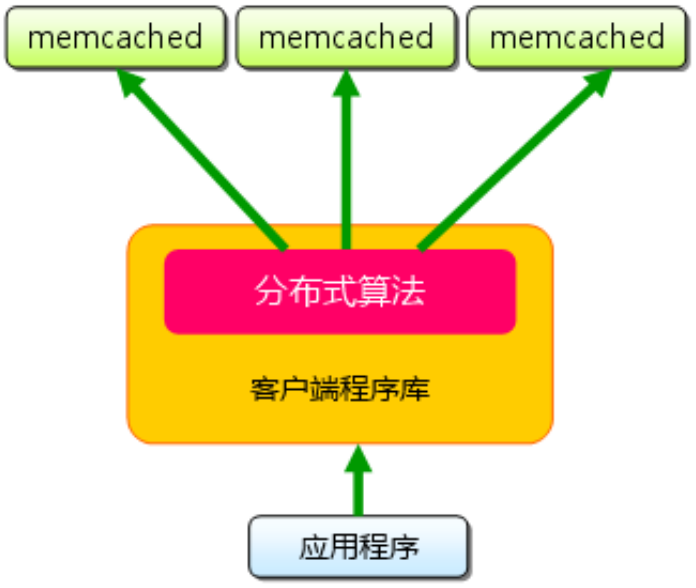
我发明的新概念，就是称不上数据库但有一些数据库的特征。可以指缓存。

MemCached

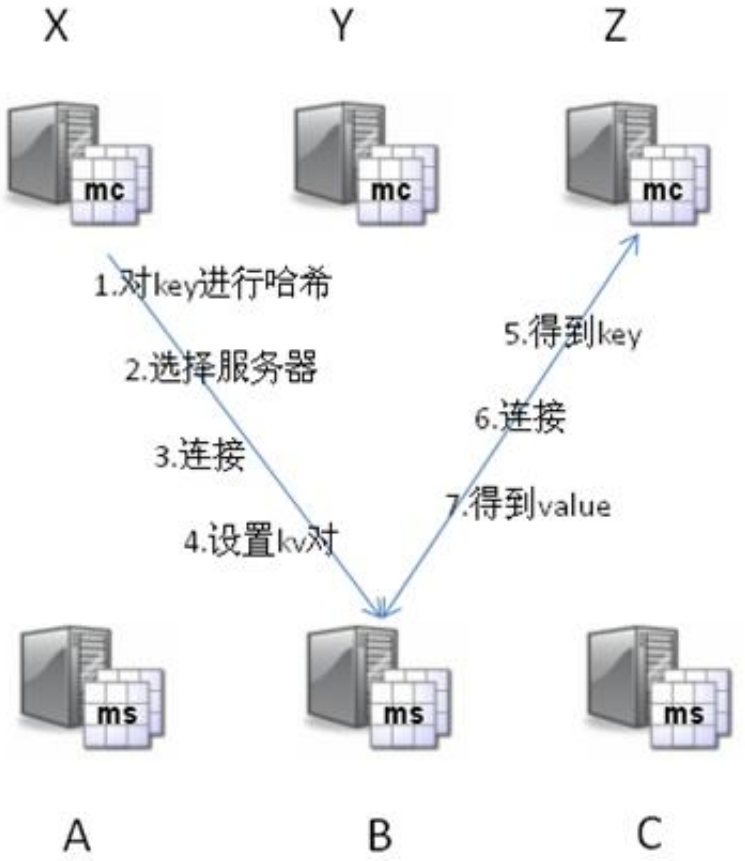
Memcached是danga.com（运营LiveJournal的技术团队）开发的一套分布式内存对象缓存系统，用于在动态系统中减少数据库 负载，提升性能。

特点

- 协议简单
- 基于libevent的事件处理
- 内置内存存储方式
- memcached不互相通信的分布式



Memcached处理的原子是每一个（key，value）对（以下简称kv对），key会通过一个hash算法转化成hash-key，便于查找、对比以及做到尽可能的散列。同时，memcached用的是一个二级散列，通过一张大hash表来维护。
Memcached有两个核心组件组成：服务端（ms）和客户端（mc），在一个memcached的查询中，mc先通过计算key的hash值来 确定kv对所处在的ms位置。当ms确定后，客户端就会发送一个查询请求给对应的ms，让它来查找确切的数据。因为这之间没有交互以及多播协议，所以 memcached交互带给网络的影响是最小化的。



内存分配

默认情况下，ms是用一个内置的叫"块分配器"的组件来分配内存的。舍弃c++标准的malloc/free的内存分配，而采用块分配器的主要目的 是为了避免内存碎片，否则操作系统要花费更多时间来查找这些逻辑上连续的内存块（实际上

是断开的）。用了块分配器，ms会轮流的对内存进行大块的分配，并 不断重用。当然由于块的大小各不相同，当数据大小和块大小不太相符的情况下，还是有可能导致内存的浪费。
同时，ms对key和data都有相应的限制，key的长度不能超过250字节，data也不能超过块大小的限制 --- 1MB。
因为mc所使用的hash算法，并不会考虑到每个ms的内存大小。理论上mc会分配概率上等量的kv对给每个ms，这样如果每个ms的内存都不太一样，那 可能会导致内存使用率的降低。所以一种替代的解决方案是，根据每个ms的内存大小，找出他们的最大公约数，然后在每个ms上开n个容量=最大公约数的 instance，这样就等于拥有了多个容量大小一样的子ms，从而提供整体的内存使用率。

缓存策略

当ms的hash表满了之后，新的插入数据会替代老的数据，更新的策略是LRU（最近最少使用），以及每个kv对的有效时限。Kv对存储有效时限是在mc端由app设置并作为参数传给ms的。
同时ms采用是偷懒替代法，ms不会开额外的进程来实时监测过时的kv对并删除，而是当且仅当，新来一个插入的数据，而此时又没有多余的空间放了，才会进行清除动作。

缓存数据库查询

现在memcached最流行的一种使用方式是缓存数据库查询，下面举一个简单例子说明：
App需要得到userid=xxx的用户信息，对应的查询语句类似：
“SELECT * FROM users WHERE userid = xxx”
App先去问cache，有没有“user:userid”（key定义可预先定义约束好）的数据，如果有，返回数据；如果没有，App会从数据库中读取数据，并调用cache的add函数，把数据加入cache中。
当取的数据需要更新，app会调用cache的update函数，来保持数据库与cache的数据同步。
从上面的例子我们也可以发现，一旦数据库的数据发现变化，我们一定要及时更新cache中的数据，来保证app读到的是同步的正确数据。当然我们可 以通过定时器方式记录下cache中数据的失效时间，时间一过就会激发事件对cache进行更新，但这之间总会有时间上的延迟，导致app可能从 cache读到脏数据，这也被称为狗洞问题。（以后我会专门描述研究这个问题）

数据冗余与故障预防

从设计角度上，memcached是没有数据冗余环节的，它本身就是一个大规模的高性能cache层，加入数据冗余所能带来的只有设计的复杂性和提高系统的开支。
当一个ms上丢失了数据之后，app还是可以从数据库中取得数据。不过更谨慎的做法是在某些ms不能正常工作时，提供额外的ms来支持cache，这样就不会因为app从cache中取不到数据而一下子给数据库带来过大的负载。
同时为了减少某台ms故障所带来的影响，可以使用“热备份”方案，就是用一台新的ms来取代有问题的ms，当然新的ms还是要用原来ms的IP地址，大不了数据重新装载一遍。
另外一种方式，就是提高你ms的节点数，然后mc会实时侦查每个节点的状态，如果发现某个节点长时间没有响应，就会从mc的可用server列表里 删除，并对server节点进行重新hash定位。当然这样也会造成的问题是，原本key存储在B上，变成存储在C上了。所以此方案本身也有其弱点，最好 能和“热备份”方案结合使用，就可以使故障造成的影响最小化。

Memcached客户端（**mc**）

Memcached客户端有各种语言的版本供大家使用，包括java，c，php，.net等等，具体可参见[memcached api page](#) [2]。
大家可以根据自己项目的需要，选择合适的客户端来集成。

缓存式的**Web**应用程序架构

有了缓存的支持，我们可以在传统的app层和db层之间加入cache层，每个app服务器都可以绑定一个mc，每次数据的读取都可以从ms中取得，如果 没有，再从db层读取。而当数据要进行更新时，除了要发送update的sql给db层，同时也要将更新的数据发给mc，让mc去更新ms中的数据。

性能测试

Memcached 写速度
平均速度: 16222 次/秒
最大速度 18799 次/秒

Memcached 读速度
平均速度: 20971 次/秒
最大速度 22497 次/秒

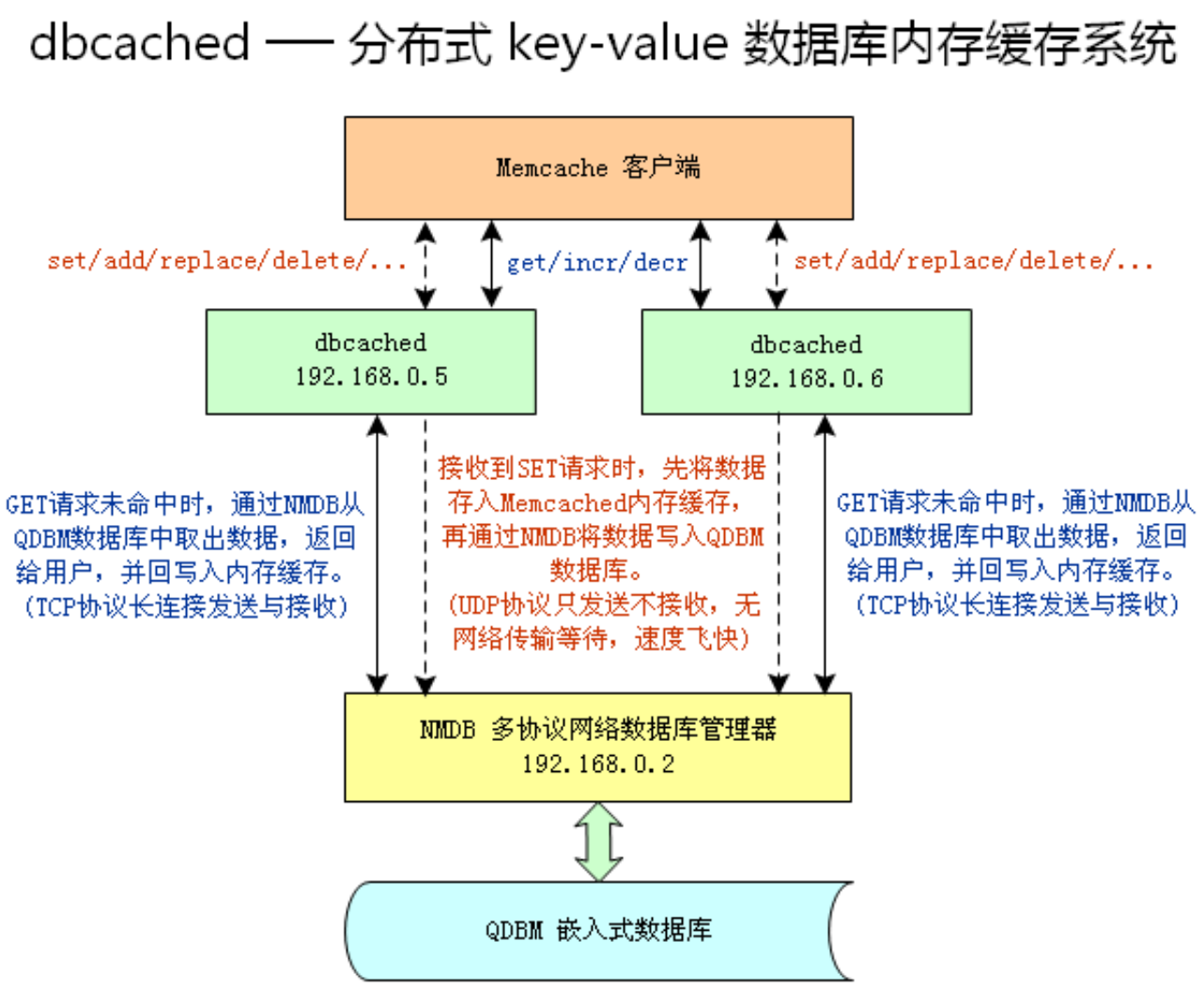
Memcachedb 写速度
平均速度: 8958 次/秒
最大速度 10480 次/秒

Memcachedb 读速度
平均速度: 6871 次/秒
最大速度 12542 次/秒

源代码级别的分析
非常好的剖析文章

dbcached

- dbcached 是一款基于 Memcached 和 NMDB 的分布式 key-value 数据库内存缓存系统。
- dbcached = Memcached + 持久化存储管理器 + NMDB 客户端接口
- Memcached 是一款高性能的，分布式的内存对象缓存系统，用于在动态应用中减少数据库负载，提升访问速度。
- NMDB 是一款多协议网络数据库(dbm类)管理器，它由内存缓存和磁盘存储两部分构成，使用 QDBM 或 Berkeley DB 作为后端数据库。
- QDBM 是一个管理数据库的例程库，它参照 GDBM 为了下述三点而被开发：更高的处理速度，更小的数据库文件大小，和更简单的API。QDBM 读写速度比 Berkeley DB 要快，详细速度比较见《Report of Benchmark Test》。



Memcached 和 **dbcached** 在功能上一样吗**?**

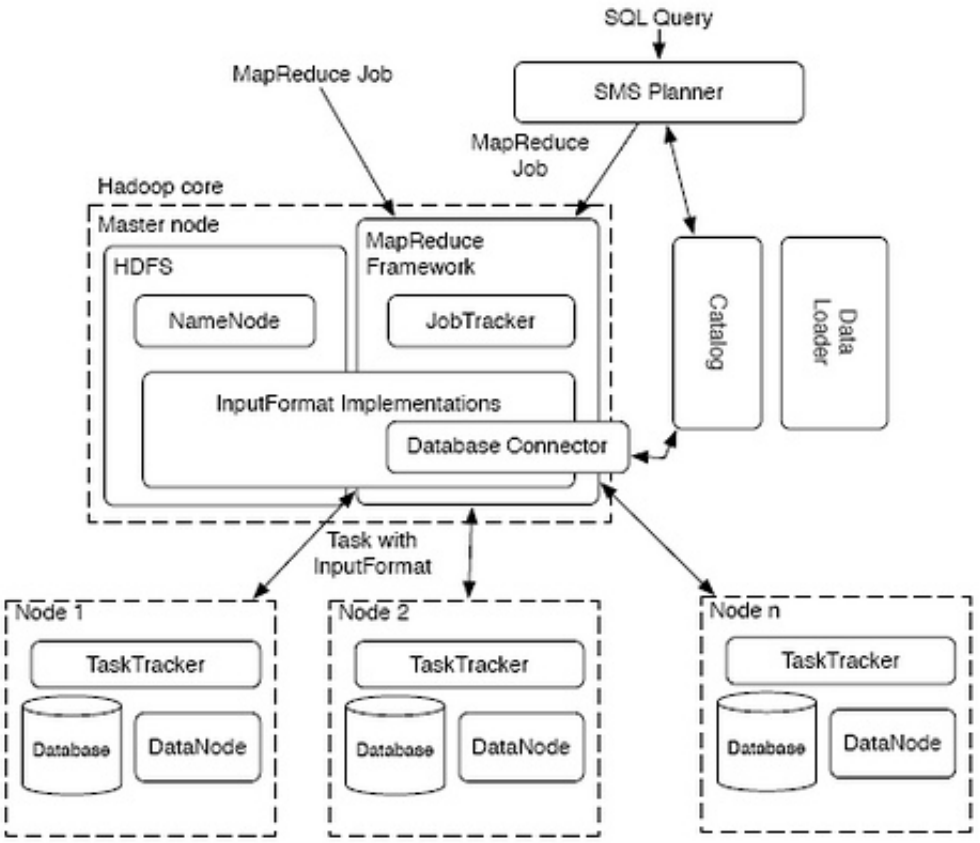
● 兼容：Memcached 能做的，dbcached 都能做。除此之外，dbcached 还将"Memcached、持久化存储管理器、NMDB 客户端接口"在一个程序中结合起来，对任何原有 Memcached 客户端来讲，dbcached 仍旧是个 Memcached 内存对象缓存系统，但是，它的数据可以持久存储到本机或其它服务器上的 QDBM 或 Berkeley DB 数据库中。
● 性能：前端 dbcached 的并发处理能力跟 Memcached 相同；后端 NMDB 跟 Memcached 一样，采用了 libevent 进行网络IO处理，拥有自己的内存缓存机制，性能不相上下。
● 写入：当"dbcached 的 Memcached 部分"接收到一个 set(add/replace/...) 请求并储存 key-value 数据到内存中后，"dbcached 持久化存储管理器"能够将 key-value 数据通过"NMDB 客户端接口"保存到 QDBM 或 Berkeley DB 数据库中。
● 速度：如果加上"-z"参数，采用 UDP 协议"只发送不接收"模式将 set(add/replace/...) 命令写入的数据传递给 NMDB 服务器端，对 Memcache 客户端写速度的影响几乎可以忽略不计。在千兆网卡、同一交换机下服务器之间的 UDP 传输丢包率微乎其微。在命中的情况下，读取数据的速度跟普通的 Memcached 无差别，速度一样快。
● 读取：当"dbcached 的 Memcached 部分"接收到一个 get(incr/decr/...) 请求后，如果"dbcached 的 Memcached 部分"查询自身的内存缓存未命中，则"dbcached 持久化存储管理器"会通过"NMDB 客户端接口"从 QDBM 或 Berkeley DB 数据库中取出数据，返回给用户，然后储存到 Memcached 内存中。如果有用户再次请求这个 key，则会直接从 Memcached 内存中返回 Value 值。
● 持久：使用 dbcached，不用担心 Memcached 服务器死机、重启而导致数据丢失。
● 变更：使用 dbcached，即使因为故障转移，添加、减少 Memcached 服务器节点而破坏了"key 信息"与对应"Memcached 服务器"的映射关系也不怕。
● 分布：dbcached 和 NMDB 既可以安装在同一台服务器上，也可以安装在不同的服务器上，多台 dbcached 服务器可以对应一台 NMDB 服务器。
● 特长：dbcached 对于"读"大于"写"的应用尤其适用。
● 其他：《dbcached 的故障转移支持、设计方向以及与 Memcachedb 的不同之处》

列存系列

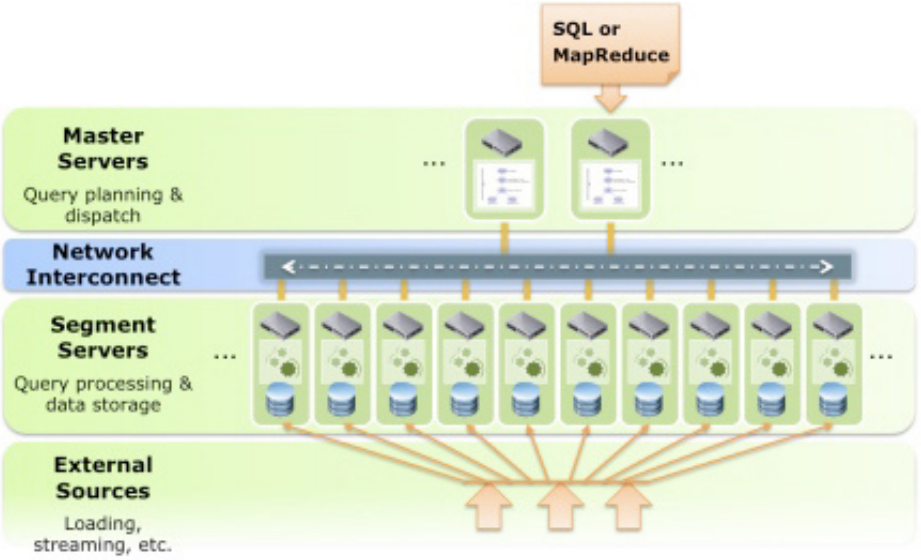
Hadoop之Hbase

Hadoop / HBase: API: **Java / any writer**, Protocol: **any write call**, Query Method: **MapReduce Java / any exec**, Replication: **HDFS Replication**, Written in: **Java**, Concurrency: ?, Misc: **Links**: 3 Books [1, 2, 3]

耶鲁大学之HadoopDB



GreenPlum



FaceBook之Cassandra

Cassandra: API: **many Thrift » languages**, Protocol: ?, Query Method: **MapReduce**, Replicaton: , Written in: **Java**, Concurrency: **eventually consistent** , Misc: like "Big-Table on Amazon Dynamo alike", initiated by Facebook, Slides » , Clients »

Cassandra是facebook开源出来的一个版本，可以认为是BigTable的一个开源版本，目前twitter和digg.com在使用。

Cassandra特点

- 灵活的schema，不需要象数据库一样预先设计schema，增加或者删除字段非常方便（on the fly）。
- 支持range查询：可以对Key进行范围查询。
- 高可用，可扩展：单点故障不影响集群服务，可线性扩展。

Cassandra的主要特点就是它不是一个数据库，而是由一堆数据库节点共同构成的一个分布式网络服务，对Cassandra的一个写操作，会 被复制到其他节点上去，对Cassandra的读操作，也会被路由到某个节点上面去读取。对于一个Cassandra群集来说，扩展性能是比较简单的事 情，只管在群集里面添加节点就可以了。我看到有文章说Facebook的Cassandra群集有超过100台服务器构成的数据库群集。

Cassandra也支持比较丰富的数据结构和功能强大的查询语言，和MongoDB比较类似，查询功能比MongoDB稍弱一些，twitter的平台架构部门领导Evan Weaver写了一篇文章介绍Cassandra：http://blog.evanweaver.com/articles/2009/07/06/up-and-running-with-cassandra/，有非常详细的介绍。

Cassandra以单个节点来衡量，其节点的并发读写性能不是特别好，有文章说评测下来Cassandra每秒大约不到1万次读写请求，我也看 到一些对这个问题进行质疑的评论，但是评价Cassandra单个节点的性能是没有意义的，真实的分布式数据库访问系统必然是n多个节点构成的系统，其并 发性能取决于整个系统的节点数量，路由效率，而不仅仅是单节点的并发负载能力。

Keyspace

Cassandra中的最大组织单元，里面包含了一系列Column family，Keyspace一般是应用程序的名称。你可以把它理解为Oracle里面的一个schema，包含了一系列的对象。

Column family（**CF**）

CF是某个特定Key的数据集合，每个CF物理上被存放在单独的文件中。从概念上看，CF有点象数据库中的Table.

Key

数据必须通过Key来访问，Cassandra允许范围查询，例如：start => '10050', :finish => '10070'

Column

在Cassandra中字段是最小的数据单元，column和value构成一个对，比如：name:"jacky"，column是name，value是jacky，每个column:value后都有一个时间戳：timestamp。

和数据库不同的是，Cassandra的一行中可以有任意多个column，而且每行的column可以是不同的。从数据库设计的角度，你可以理解 为表上有两个字段，第一个是Key，第二个是长文本类型，用来存放很多的column。这也是为什么说Cassandra具备非常灵活schema的原 因。

Super column

Super column是一种特殊的column，里面可以存放任意多个普通的column。而且一个CF中同样可以有任意多个Super column，一个CF只能定义使用Column或者Super column，不能混用。下面是Super column的一个例子，homeAddress这个Super column有三个字段：分别是street，city和zip： homeAddress: {street: "binjiang road",city: "hangzhou",zip: "310052",}

Sorting

不同于数据库可以通过Order by定义排序规则，Cassandra取出的数据顺序是总是一定的，数据保存时已经按照定义的规则存放，所以取出来的顺序已经确定了，这是一个巨大的性能优势。有意思的是，Cassandra按照column name而不是column value来进行排序，它 定义了以下几种选项：BytesType, UTF8Type, LexicalUUIDType, TimeUUIDType, AsciiType, 和LongType，用来定义如何按照column name来排序。实际上，就是把column name识别成为不同的类型，以此来达到灵活排序的目的。UTF8Type是把column name转换为UTF8编码来进行排序，LongType转换成为64位long型，TimeUUIDType是按照基于时间的UUID来排序。例如：

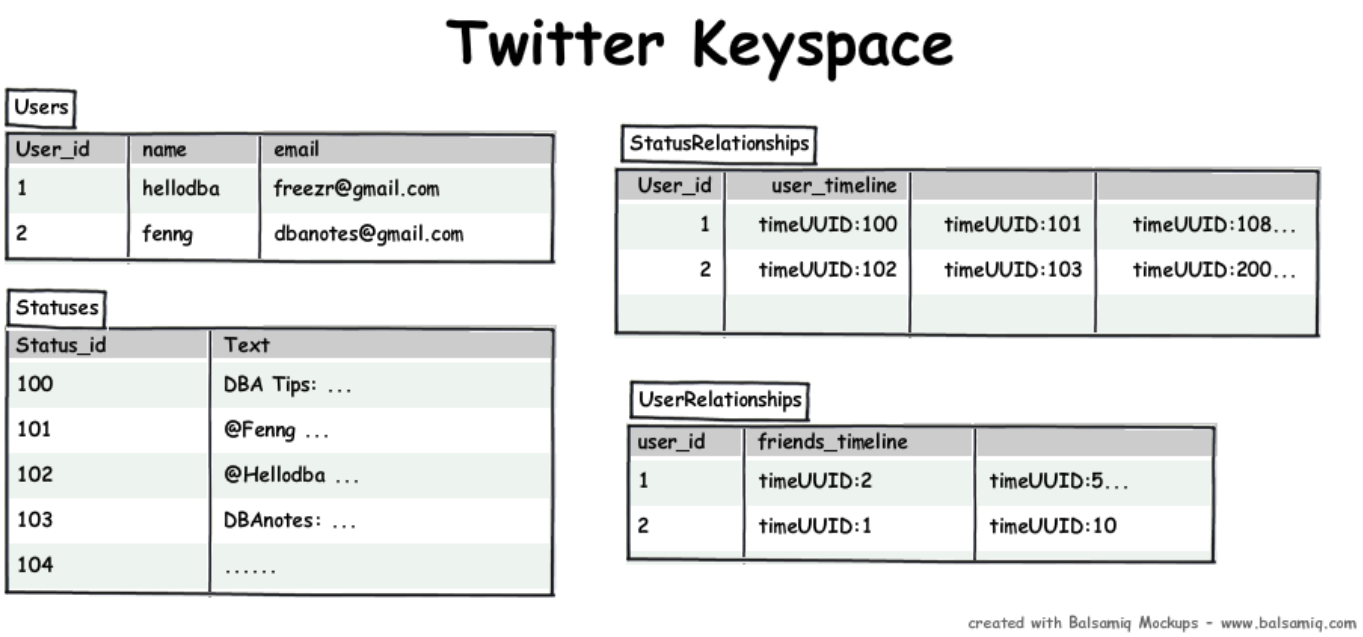
Column name按照LongType排序：
{name: 3, value: "jacky"},
{name: 123, value: "hellodba"},
{name: 976, value: "Cassandra"},
{name: 832416, value: "bigtable"}

Column name按照UTF8Type排序：
{name: 123, value: "hellodba"},
{name: 3, value: "jacky"},
{name: 832416, value: "bigtable"}
{name: 976, value: "Cassandra"}

下面我们看twitter的Schema：
<Keyspace Name="Twitter">
<ColumnFamily CompareWith="UTF8Type" Name="Statuses" />
<ColumnFamily CompareWith="UTF8Type" Name="StatusAudits" />
<ColumnFamily CompareWith="UTF8Type" Name="StatusRelationships"
CompareSubcolumnsWith="TimeUUIDType" ColumnType="Super" />
<ColumnFamily CompareWith="UTF8Type" Name="Users" />
<ColumnFamily CompareWith="UTF8Type" Name="UserRelationships"
CompareSubcolumnsWith="TimeUUIDType" ColumnType="Super" />
</Keyspace>

我们看到一个叫Twitter的keyspace，包含若干个CF，其中StatusRelationships和 UserRelationships被定义为包含Super column的CF，CompareWith定义了column的排序规则，CompareSubcolumnsWith定义了subcolumn的排序 规则，这里使用了两种：TimeUUIDType和UTF8Type。我们没有看到任何有关column的定义，这意味着column是可以灵活变更的。

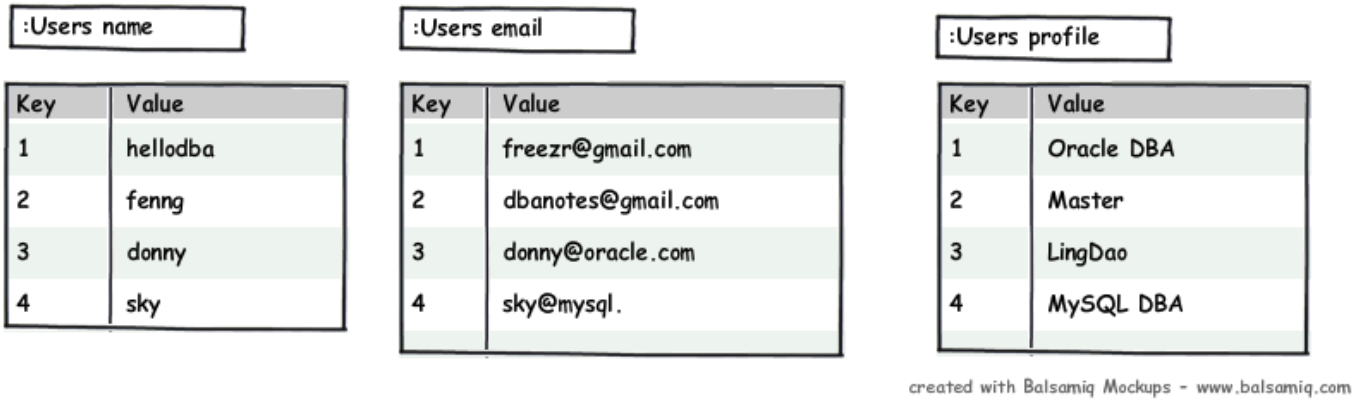
为了方便大家理解，我会尝试着用关系型数据库的建模方法去描述Twitter的Schema，但千万不要误认为这就是Cassandra的数据模型，对于Cassandra来说，每一行的colunn都可以是任意的，而不是象数据库一样需要在建表时就创建好。



Users CF记录用户的信息，Statuses CF记录tweets的内容，StatusRelationships CF记录用户看到的tweets，UserRelationships CF记录用户看到的followers。我们注意到排序方式是TimeUUIDType，这个类型是按照时间进行排序的UUID字段，column name是用UUID函数产生（这个函数返回了一个UUID，这个UUID反映了当前的时间，可以根据这个UUID来排序，有点类似于timestamp 一样），所以得到结果是按照时间来排序的。使用过twitter的人都知道，你总是可以看到自己最新的tweets或者最新的friends.

存储

Cassandra是基于列存储的(Bigtable也是一样)，这个和基于列的数据库是一个道理。



API

下面是数据库，Bigtable和Cassandra API的对比： Relational SELECT `column` FROM `database`.`table` WHERE `id` = key;
BigTable table.get(key, "column_family:column")
Cassandra: standard model keyspace.get("column_family", key, "column")
Cassandra: super column model keyspace.get("column_family", key, "super_column", "column")

我对Cassandra数据模型的理解：

1.column name存放真正的值，而value是空。因为Cassandra是按照column name排序，而且是按列存储的，所以往往利用column name存放真正的值，而value部分则是空。例如：“jacky”:“null”，“fenng”:”null”

2.Super column可以看作是一个索引，有点象关系型数据库中的外键，利用super column可以实现快速定位，因为它可以返回一堆column，而且是排好序的。

3.排序在定义时就确定了，取出的数据肯定是按照确定的顺序排列的，这是一个巨大的性能优势。

4. 非常灵活的schema，column可以灵活定义。实际上，colume name在很多情况下，就是value（是不是有点绕）。

5.每个column后面的timestamp，我并没有找到明确的说明，我猜测可能是数据多版本，或者是底层清理数据时需要的信息。

最后说说架构，我认为架构的核心就是有所取舍，不管是CAP还是BASE，讲的都是这个原则。架构之美在于没有任何一种架构可以完美的解决各种问题，数据库和NoSQL都有其应用场景，我们要做的就是为自己找到合适的架构。

Hypertable

Hypertable: (can you help?) Open-Source Google BigTable alike.

它是搜索引擎公司Zvents根据Google的9位研究人员在2006年发表的一篇论文《Bigtable：结构化数据的分布存储系统》 开发的一款开源分布式数据储存系统。Hypertable是按照1000节点比例设计，以 C++撰写，可架在 HDFS 和 KFS 上。尽管还在初期阶段，但已有不错的效能：写入 28M 列的资料，各节点写入速率可达7MB/s，读取速率可达 1M cells/s。Hypertable目前一直没有太多高负载和大存储的应用实例，但是最近，Hypertable项目得到了百度的赞助支持，相信其会有更好的发展。

Google之BigTable

研究Google的产品总是感激Google给了自己那么多方便，真心喜欢之。

Google AppEngine Datastore 是在BigTable之上建造出来的，是Google的内部存储系统，用于处理结构化数据。AppEngine Datastore其自身及其内部都不是直接访问BigTable的实现机制，可被视为BigTable之上的一个简单接口。

AppEngine Datastore所支持的项目的数据类型要比SimpleDB丰富得多，也包括了包含在一个项目内的数据集合的列表型。

如果你打算在Google AppEngine之内建造应用的话，几乎可以肯定要用到这个数据存储。然而，不像SimpleDB，使用谷歌网络服务平台之外的应用，你并不能并发地与AppEngine Datastore进行接口 (或通过BigTable)。

Yahoo之PNUTS

Yahoo!的PNUTS是一个分布式的数据存储平台，它是Yahoo!云计算平台重要的一部分。它的上层产品通常也称为Sherpa。按照官方的 描述，”PNUTS, a massively parallel and geographically distributed database system for Yahoo!’s web applications.” PNUTS显然就深谙CAP之道，考虑到大部分web应用对一致性并不要求非常严格，在设计上放弃了对强一致性的追求。代替的是追求更高的 availability，容错，更快速的响应调用请求等。

特点

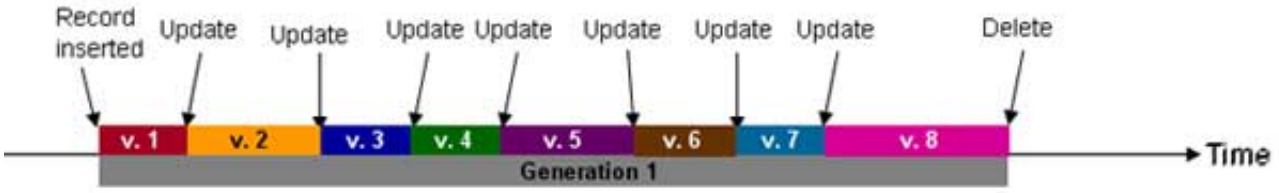
- 地理分布式，分布在全球多个数据中心。由于大部分Web应用都对响应时间要求高，因此最好服务器部署在离用户最近的本地机房。
- 可扩展，记录数可支持从几万条到几亿条。数据容量增加不会影响性能。
- schema-free，即非固定表结构。实际使用key/value存储的，一条记录的多个字段实际是用json方式合并存在value中。因此delete和update必须指定primary key。但也支持批量查询。
- 高可用性及容错。从单个存储节点到整个数据中心不可用都不会影响前端Web访问。

- 适合存相对小型的记录，不适合存储大文件，流媒体等。
- 弱一致性保证。

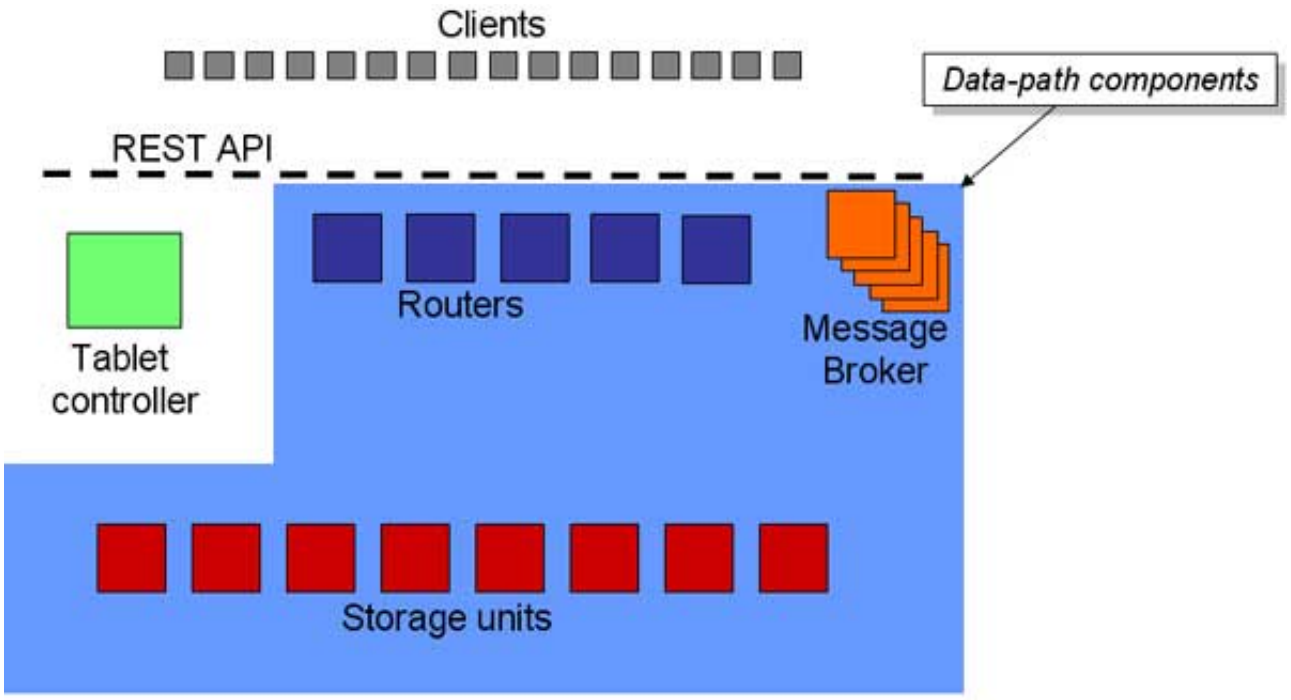
PNUTS实现

Record-level mastering 记录级别主节点

每一条记录都有一个主记录。比如一个印度的用户保存的记录master在印度机房，通常修改都会调用印度。其他地方如美国用户看这个用户的资料调用 的是美国数据中心的资料，有可能取到的是旧版的数据。非master机房也可对记录进行修改，但需要master来统一管理。每行数据都有自己的版本控 制，如下图所示。



PNUTS的结构



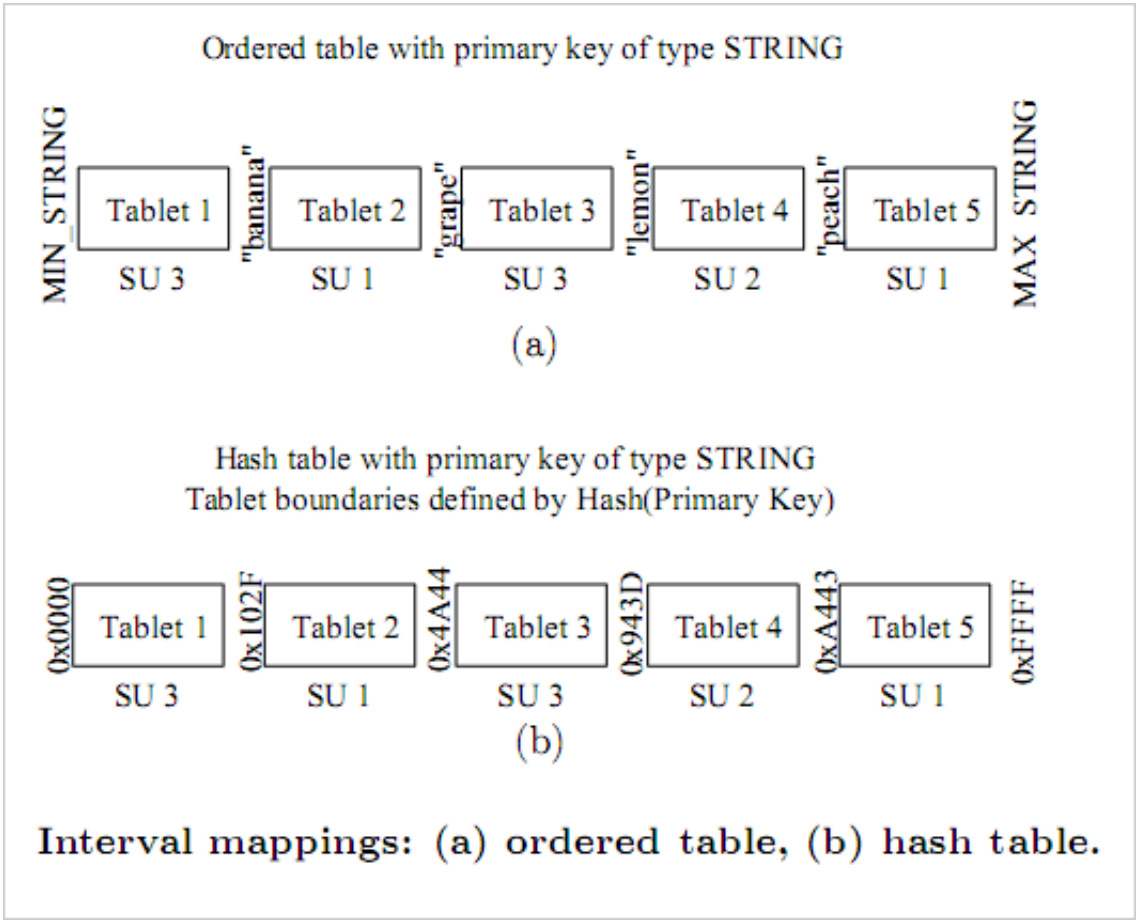
每个数据中心的PNUTS结构由四部分构成
Storage Units (SU) 存储单元
物理的存储服务器，每个存储服务器上面含有多个tablets，tablets是PNUTS上的基本存储单元。一 个tablets是一个yahoo内部格式的hash table的文件(hash table)或是一个MySQL innodb表(ordered table)。一个Tablet通常为几百M。一个SU上通常会存在几百个tablets。
Routers
每个tablets在哪个SU上是通过查询router获得。一个数据中心内router通常可由两台双机备份的单元提供。

Tablet Controller
router的位置只是个内存快照，实际的位置由Tablet Controller单元决定。
Message Broker

与远程数据的同步是由YMB提供，它是一个pub/sub的异步消息订阅系统。

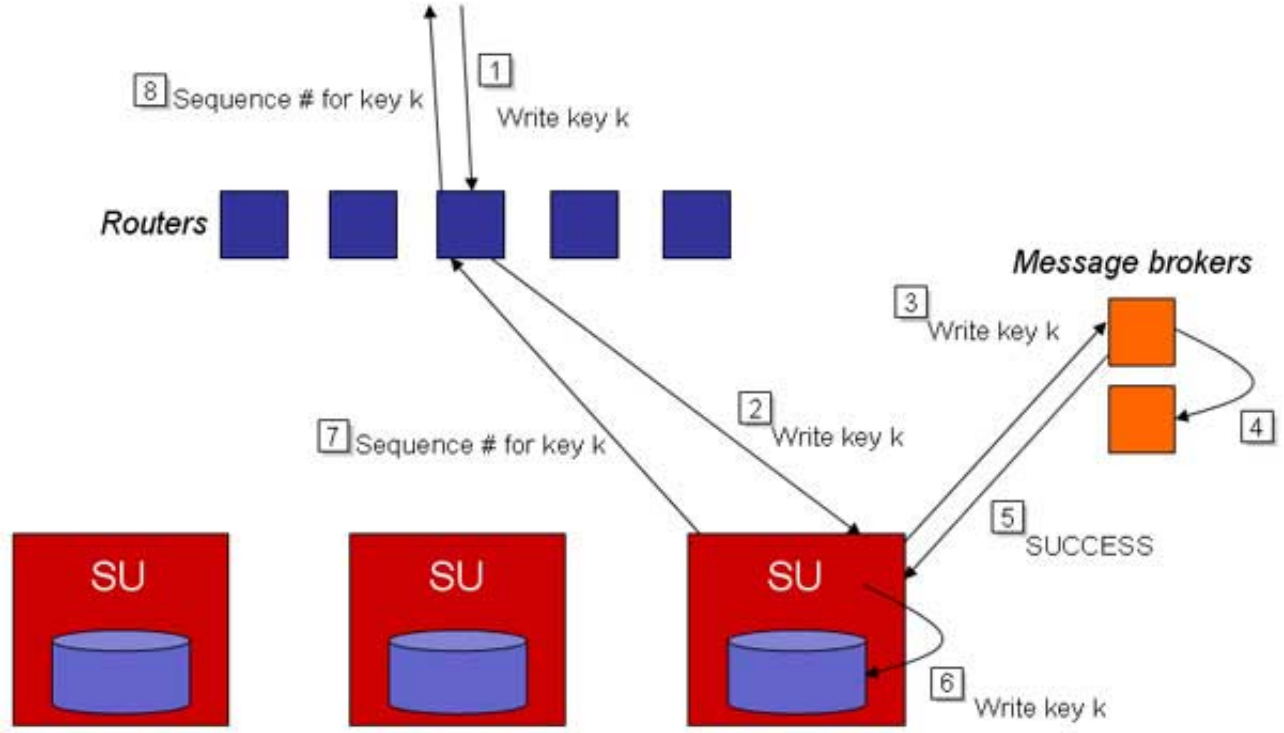
Tablets寻址与切分

存储分hash和ordered data store。



以hash为例介绍，先对所有的tablets按hash值分片，比如1-10,000属于tablets 1, 10,000到20,000属于tablets 2，依此类推分配完所有的hash范围。一个大型的IDC通常会存在100万以下的tablets, 1,000台左右的SU。tablets属于哪个SU由routers全部加载到内存里面，因此router访问速度极快，通常不会成为瓶颈。按照官方的 说法，系统的瓶颈只存在磁盘文件hash file访问上。
当某个SU访问量过大，则可将SU中部分tablets移到相对空闲的SU，并修改tablet controller的偏移记录。router定位tablet失效之后会自动通过tablet controller重新加载到内存。所以切分也相对容易实现。
Tim也曾经用MySQL实现过类似大规模存储的系统，当时的做法是把每条记录的key属于哪个SU的信息保存到 一个字典里面，好处是切分可以获得更大的灵活性，可以动态增加新的tablets,而不需要切分旧的tablets。但缺点就是字典没法像router这 样，可以高效的全部加载到内存中。所以比较而言，在实际的应用中，按段分片会更简单，且已经足够使用。

Write调用示意图



PNUTS感悟

2006年Greg Linden就说I want a big, virtual database

What I want is a robust, high performance virtual relational database that runs transparently over a cluster, nodes dropping in an out of service at will, read-write replication and data migration all done automatically.

I want to be able to install a database on a server cloud and use it like it was all running on one machine.

详细资料：
http://timyang.net/architecture/yahoo-pnuts/

微软之**SQL**数据服务

SQL数据服务 是微软 Azure 网 络服务平台的一部分。该SDS服务也是处于测试阶段，因此也是免费的，但对数据库大小有限制。 SQL数据服务其自身实际上是一项处在许多SQL服务器之上的应用，这些SQL服务器组成了SDS平台底层的数据存储。你不需要访问到它们，虽然底层的数 据库可能是关系式的；SDS是一个键/值型仓储，正如我们迄今所讨论过的其它平台一样。

微软看起来不同于前三个供应商，因为虽然键/值存储对于可扩性���言非常棒，相对于RDBMS，在数据管理上却很困难。微软的方案似乎是入木三分，在实现可扩性和分布机制的同时，随着时间的推移，不断增加特性，在键/值存储和关系数据库平台的鸿沟之间搭起一座桥梁。

非云服务竞争者

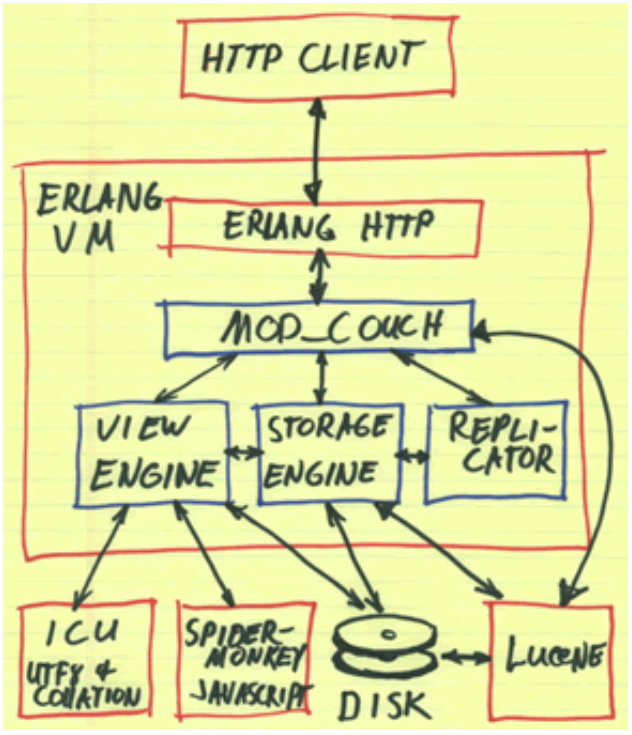
在云之外，也有一些可以独立安装的键/值数据库软件产品。大部分都还很年轻，不是alpha版就是beta版，但大都是开源的；通过看看它的代码，比起在非开源供应商那里，你也许更能意识到潜在的问题和限制。

文档存储

CouchDB

CouchDB: API: **JSON**, Protocol: **REST**, Query Method: **MapReduceR of JavaScript Funcs**, Replication: **Master Master**, Written in: **Erlang**, Concurrency: **MVCC**, **Misc**:
Links: 3 CouchDB books », Couch Lounge » (partitioning / clusering), ...

它是Apache社区基于 Erlang/OTP 构建的高性能、分布式容错非关系型数据库系统（NRDBMS）。它充分利用 Erlang 本身所提供的高并发、分布式容错基础平台，并且参考 Lotus Notes 数据库实现，采用简单的文档数据类型（document-oriented）。在其内部，文档数据均以 JSON 格式存储。对外，则通过基于 HTTP 的 REST 协议实现接口，可以用十几种语言进行自由操作。



CouchDB一种半结构化面向文档的分布式，高容错的数据库系统，其提供RESTFul HTTP/JSON接口。其拥有MVCC特性，用户可以通过自定义Map/Reduce函数生成对应的View。

在CouchDB中，数据是以JSON字符的方式存储在文件中。

特性

- RESTFul API：HTTP GET/PUT/POST/DELETE + JSON
- 基于文档存储，数据之间没有关系范式要求
- 每个数据库对应单个个文件(以JSON保存),Hot backup
- MVCC（Multi-Version-Concurrency-Control），读写均不锁定数据库
- 用户自定义View
- 内建备份机制
- 支持附件
- 使用Erlang开发（更多的特性）

应用场景 在我们的生活中，有很多document，比如信件，账单，笔记等，他们只是简单的信息，没有关系的需求，我们可能仅仅需要存储这些数据。 这样的情况下，CouchDB应该是很好的选择。当然其他使用关系型数据库的环境，也可以使用CouchDB来解决。

根据CouchDB的特性，在某些偶 尔连接网络的应用中，我们可以用CouchDB暂存数据，随后进行同步。也可以在Cloud环境中，作为分布式的数据存储。CouchDB提供给予 HTTP的API，这样所有的常见语言都可以使用CouchDB。

使用CouchDB，意味着我们不需要在像使用RMDBS一样，在设计应用前首先设计负责数据Table。我们的开发更加快速，灵活。

详细参见：
http://www.javaeye.com/topic/319839

Riak

Riak: API: **JSON**, Protocol: **REST**, Query Method: **MapReduce term matching** , Scaling: **Multiple Masters**; Written in: **Erlang**, Concurrency: **eventually consistent** (stronger then MVCC via Vector Clocks), **Misc:** ... **Links**: talk »,

MongoDB

MongoDB: API: **BSON**, Protocol: **lots of langs**, Query Method: **dynamic object-based language**, Replication: **Master Slave**, Written in: **C++**,Concurrency: **Update in Place**. **Misc:** ... **Links**: Talk »,

MongoDB是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。他支持的数据结构非常松散，是 类似json的bjson格式，因此可以存储比较复杂的数据类型。Mongo最大的特点是他支持的查询语言非常强大，其语法有点类似于面向对象的查询语 言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且还支持对数据建立索引。

Mongo主要解决的是海量数据的访问效率问题，根据官方的文档，当数据量达到50GB以上的时候，Mongo的数据库访问速度是MySQL的 10倍以上。Mongo的并发读写效率不是特别出色，根据官方提供的性能测试表明，大约每秒可以处理0.5万－1.5次读写请求。对于Mongo的并发读 写性能，我（robbin）也打算有空的时候好好测试一下。

因为Mongo主要是支持海量数据存储的，所以Mongo还自带了一个出色的分布式文件系统GridFS，可以支持海量的数据存储，但我也看到有些评论认为GridFS性能不佳，这一点还是有待亲自做点测试来验证了。

最后由于Mongo可以支持复杂的数据结构，而且带有强大的数据查询功能，因此非常受到欢迎，很多项目都考虑用MongoDB来替代MySQL来实现不是特别复杂的Web应用，比方说why we migrated from MySQL to MongoDB就是一个真实的从MySQL迁移到MongoDB的案例，由于数据量实在太大，所以迁移到了Mongo上面，数据查询的速度得到了非常显著的提升。

MongoDB也有一个ruby的项目MongoMapper，是模仿Merb的DataMapper编写的MongoDB的接口，使用起来非常简单，几乎和DataMapper一模一样，功能非常强大易用。

Terrastore

Terrastore: API: **Java & http**, Protocol: **http**, Language: **Java**, Querying: **Range queries, Predicates**, Replication: **Partitioned with consistent hashing**, Consistency: **Per-record strict consistency**, Misc: Based on Terracotta

ThruDB

ThruDB: (please help provide more facts!) Uses Apache Thrift to integrate multiple backend databases as BerkeleyDB, Disk, MySQL, S3.

Key Value / Tuple 存储

Amazon之SimpleDB

Amazon SimpleDB: **Misc**: not open source, Book »
SimpleDB 是一个亚马逊网络服务平台的一个面向属性的键/值数据库。SimpleDB仍处于公众测试阶段；当前，用户能在线注册其"免费"版 --免费的意思是说直到超出使用限制为止。

SimpleDB有几方面的限制。首先，一次查询最多只能执行5秒钟。其次，除了字符串类型，别无其它数据类型。一切

都以字符串形式被存储、获取和 比较，因此除非你把所有日期都转为ISO8601，否则日期比较将不起作用。第三，任何字符串长度都不能超过1024字节，这限制了你在一个属性中能存储 的文本的大小（比如说产品描述等）。不过，由于该模式动态灵活，你可以通过追加"产品描述1"、"产品描述2"等来绕过这类限制。一个项目最多可以有 256个属性。由于处在测试阶段，SimpleDB的域不能大于10GB，整个库容量则不能超过1TB。

SimpleDB的一项关键特性是它使用一种最终一致性模型。 这个一致性模型对并发性很有好处，但意味着在你改变了项目属性之后，那些改变有可能不能立即反映到随后的读操作上。尽管这种情况实际发生的几率很低，你也 得有所考虑。比如说，在你的演出订票系统里，你不会想把最后一张音乐会门票卖给5个人，因为在售出时你的数据是不一致的。

Chordless

Chordless: API: **Java & simple RPC to vals**, Protocol: **internal**, Query Method: **M/R inside value objects**, Scaling: **every node is master for its slice of namespace**, Written in: **Java**, Concurrency: **serializable transaction isolation**, **Links**:

Redis

Redis : (please help provide more facts!) API: **Tons of languages**, Written in: **C**, Concurrency: **in memory** and saves asynchronous disk after a defined time. Append only mode available. Different kinds of fsync policies. Replication: **Master / Slave**,

Redis是一个很新的项目，刚刚发布了1.0版本。Redis本质上是一个Key-Value类型的内存数据库，很像memcached，整个数据库统 统加载在内存当中进行操作，定期通过异步操作把数据库数据flush到硬盘上进行保存。因为是纯内存操作，Redis的性能非常出色，每秒可以处理超过 10万次读写操作，是我知道的性能最快的Key-Value DB。

Redis的出色之处不仅仅是性能，Redis最大的魅力是支持保存List链表和Set集合的数据结构，而且还支持对List进行各种操作，例 如从List两端push和pop数据，取List区间，排序等等，对Set支持各种集合的并集交集操作，此外单个value的最大限制是1GB，不像 memcached只能保存1MB的数据，因此Redis可以用来实现很多有用的功能，比方说用他的List来做FIFO双向链表，实现一个轻量级的高性 能消息队列服务，用他的Set可以做高性能的tag系统等等。另外Redis也可以对存入的Key-Value设置expire时间，因此也可以被当作一 个功能加强版的memcached来用。

Redis的主要缺点是数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，并且它没有原生的可扩展机制，不具有scale（可扩展） 能力，要依赖客户端来实现分布式读写，因此Redis适合的场景主要局限在较小数据量的高性能操作和运算上。目前使用Redis的网站有 github，Engine Yard。

Scalaris

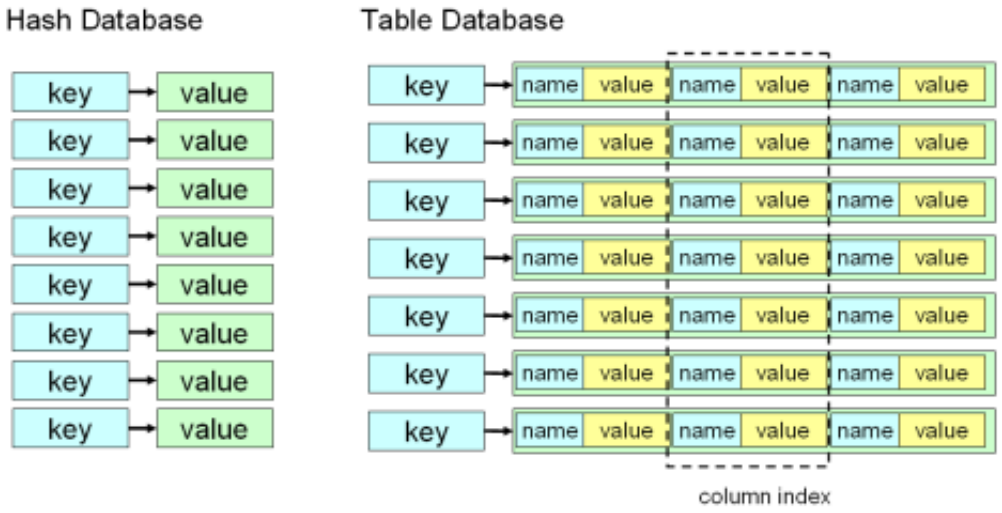
Scalaris: (please help provide more facts!) Written in: **Erlang**, Replication: **Strong consistency over replicas**, Concurrency: **non blocking Paxos**.

Tokyo cabinet / Tyrant

Tokyo Cabinet / Tyrant: **Links**: nice talk », slides », Misc: **Kyoto** Cabinet »

它是日本最大的SNS社交网站mixi.jp开发的 Tokyo Cabinet key-value数据库网络接口。它拥有Memcached兼容协议，也可以通过HTTP协议进行数据交换。对任何原有Memcached客户端来讲， 可以将Tokyo Tyrant看成是一个Memcached，但是，它的数据是可以持久存储的。Tokyo Tyrant 具有故障转移、日志文件体积小、大数据量下表现出色等优势，详见：http://blog.s135.com/post/362.htm

Tokyo Cabinet 2009年1月18日发布的新版本（Version 1.4.0）已经实现 Table Database，将key-value数据库又扩展了一步，有了MySQL等关系型数据库的表和字段的概念，相信不久的将来，Tokyo Tyrant 也将支持这一功能。值得期待。



TC除了支持Key-Value存储之外，还支持保存Hashtable数据类型，因此很像一个简单的数据库表，并且还支持基于column的条 件查询，分页查询和排序功能，基本上相当于支持单表的基础查询功能了，所以可以简单的替代关系数据库的很多操作，这也是TC受到大家欢迎的主要原因之一， 有一个Ruby的项目miyazakiresistance将TT的hashtable的操作封装成和ActiveRecord一样的操作，用起来非常爽。

TC/TT在mixi的实际应用当中，存储了2000万条以上的数据，同时支撑了上万个并发连接，是一个久经考验的项目。TC在保证了极高的并发 读写性能的同时，具有可靠的数据持久化机制，同时还支持类似关系数据库表结构的hashtable以及简单的条件，分页和排序操作，是一个很棒的 NoSQL数据库。

TC的主要缺点是在数据量达到上亿级别以后，并发写数据性能会大幅度下降，NoSQL: If Only It Was That Easy 提到，他们发现在TC里面插入1.6亿条2-20KB数据的时候，写入性能开始急剧下降。看来是当数据量上亿条的时候，TC性能开始大幅度下降，从TC作者自己提供的mixi数据来看，至少上千万条数据量的时候还没有遇到这么明显的写入性能瓶颈。

这个是Tim Yang做的一个Memcached，Redis和Tokyo Tyrant的简单的性能评测，仅供参考

CT.M

GT.M: API: **M, C, Python, Perl**, Protocol: **native, inprocess C**, Misc: Wrappers: **M/DB for SimpleDB compatible HTTP** », **MDB:X** for XML », **PIP** for mapping to tables for SQL », Features: Small footprint (17MB), Terabyte Scalability, Unicode support, Database encryption, Secure, ACID transactions (single node), eventual consistency (replication), License: AGPL v3 on x86 GNU/Linux, **Links**: Slides »,

Scalien

Scalien: API / Protocol: **http** (text, html, JSON)**, C, C++, Python**, Concurrency: **Paxos**.

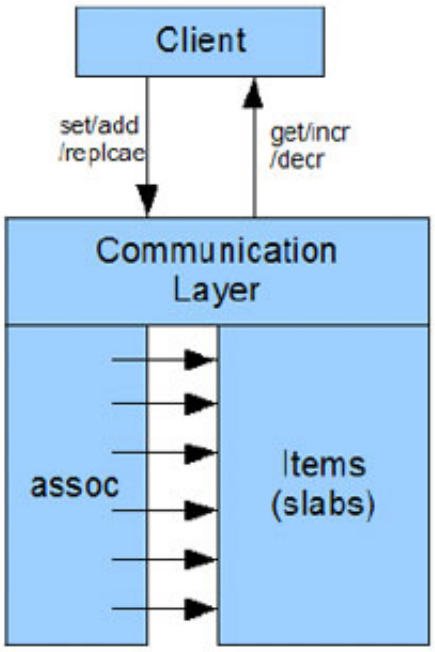
Berkley DB

Berkley DB: API: **Many languages**, Written in: **C**, Replication: **Master / Slave**, Concurrency: **MVCC**, License: **Sleepycat**, **BerkleyDB Java Edition**: API: **Java**, Written in: **Java**, Replication: **Master / Slave**, Concurrency: **serializable transaction isolation**, License: **Sleepycat**

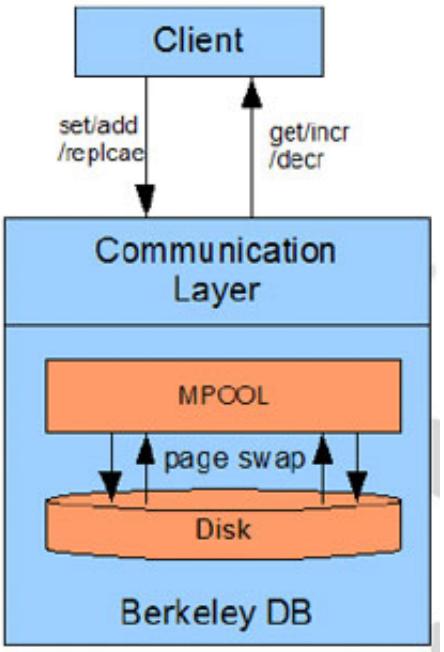
MemcacheDB

MemcacheDB: API: Memcache protocol (get, set, add, replace, etc.), Written in: **C**, Data Model: **Blob**, Misc: Is Memcached writing to BerkleyDB.

它是新浪互动社区事业部为在Memcached基础上，增加Berkeley DB存储层而开发一款支持高并发的分布式持久存储系统，对任何原有Memcached客户端来讲，它仍旧是个Memcached，但是，它的数据是可以持久存储的。



Memcached Picture



Memcachedb Picture

Mnesia

Mnesia: (ErlangDB »)

LightCloud

LightCloud: (based on Tokyo Tyrant)

HamsterDB

HamsterDB: (embedded solution) ACID Compliance, Lock Free Architecture (transactions fail on conflict rather than block), Transaction logging & fail recovery (redo logs), In Memory support – can be used as a non-persisted cache, B+ Trees – supported [Source: Tony Bain »]

Flare

TC是日本第一大SNS网站mixi开发的，而Flare是日本第二大SNS网站green.jp开发的，有意思吧。Flare简单的说就是给 TC添加了scale功能。他替换掉了TT部分，自己另外给TC写了网络服务器，Flare的主要特点就是支持scale能力，他在网络服务端之前添加了 一个node server，来管理后端的多个服务器节点，因此可以动态添加数据库服务节点，删除服务器节点，也支持failover。如果你的使用场景必须要让TC可 以scale，那么可以考虑flare。

flare唯一的缺点就是他只支持memcached协议，因此当你使用flare的时候，就不能使用TC的table数据结构了，只能使用TC的key-value数据结构存储。

最终一致性Key Value存储

Amazon之Dynamo

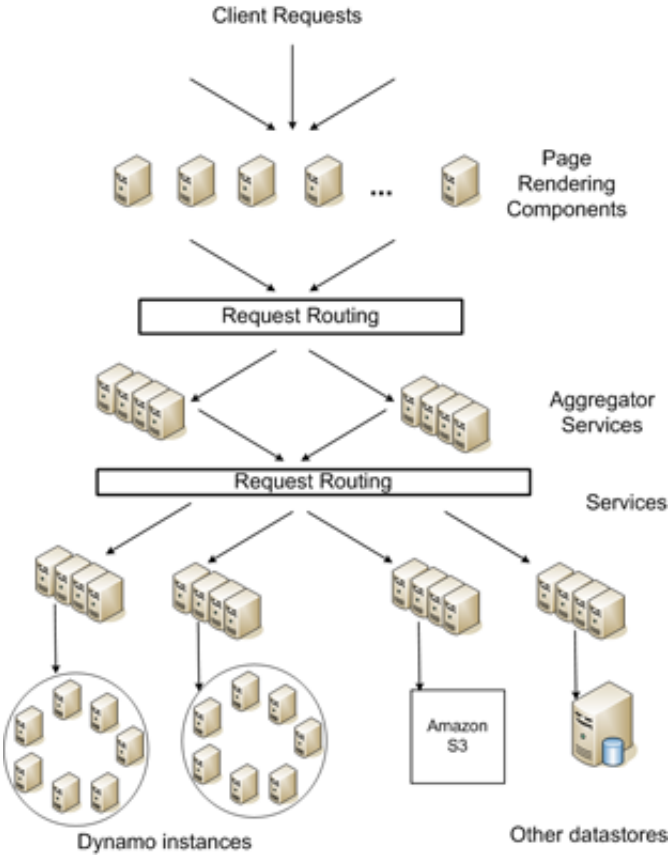
Amazon Dynamo: **Misc**: not open source (see KAI below)

功能特色

- 高可用
- 可扩展

- 总是可写
- 可以根据应用类型优化(可用性，容错性，高效性配置)

架构特色



- 完全的分布式
- 去中心化(人工管理工作很小)
- Key 唯一代表一个数据对象，对该数据对象的读写操通过 Key 来完成.
- 通常是一台自带硬盘的主机。每个节点有三个 Java 写的组件：请求协调器(request coordination)、成员与失败检测、本地持久引擎(local persistence engine)
- 数据分区并用改进的一致性哈希(consistent hashing)方式进行复制，利用数据对象的版本化实现一致性。复制时因为更新产生的一致性问题的维护采取类似 quorum 的机制以及去中心化的复制同步协议。
- 每个实例由一组节点组成，从应用的角度看，实例提供 IO 能力。一个实例上的节点可能位于不同的数据中心内，这样一个数据中心出问题也不会导致数据丢失。

BeansDB

简介

BeansDB 是一个主要针对大数据量、高可用性的分布式KeyValue存储系统，采用HashTree和简化的版本号来快速同步保证最终一致性（弱），一个简化版的Dynamo。
它采用类似memcached的去中心化结构，在客户端实现数据路由。目前只提供了Python版本的客户端，其它语言的客户端可以由memcached的客户端稍加改造得到。
Google Group: http://groups.google.com/group/beandb/

更新

2009.12.29 第一个公开版本 0.3

特性

- 高可用：通过多个可读写的用于备份实现高可用
- 最终一致性：通过哈希树实现快速完整数据同步（短时间内数据可能不一致）

- 容易扩展：可以在不中断服务的情况下进行容量扩展。
- 高性能：异步IO和高性能的KeyValue数据TokyoCabinet 可配置的
- 可用性和一致性：通过N,W,R进行配置 简单协议：Memcache兼容协议，大量可用客户端

性能

在小数据集上，它跟**memcached**一样快： **# memstorm -s localhost:7900 -n 1000**

```
Num of Records       : 10000
Non-Blocking IO      : 0
TCP No-Delay         : 0

Successful   [SET]   : 10000
Failed       [SET]   : 0
Total Time   [SET]   : 0.45493s
Average Time [SET]   : 0.00005s

Successful   [GET]   : 10000
Failed       [GET]   : 0
Total Time   [GET]   : 0.28609s
Average Time [GET]   : 0.00003s
```

实际部署情况下的性能（客户端测量）： **􀂄 服务器 请求数 评价时间(ms) 中位数(ms) 99% (ms) 99.9%(ms)**

```
&#x100084; get A:7900 n=151398, avg=8.89, med=5.94, 99%=115.5, 99.9%=310.2
&#x100084; get B:7900 n=100054, avg=6.84, med=0.40, 99%=138.5, 99.9%=483.0
&#x100084; get C:7900 n=151250, avg=7.42, med=5.34, 99%=55.2, 99.9%=156.7
&#x100084; get D:7900 n=150677, avg=7.63, med=5.09, 99%=97.7, 99.9%=284.7
&#x100084; get E:7900 n=3822,   avg=3.07, med=0.18, 99%=44.3, 99.9%=170.0
&#x100084; get F:7900 n=249973, avg=8.29, med=6.36, 99%=46.8, 99.9%=241.5
&#x100084; set A:7900 n=10177, avg=18.53, med=12.78,99%=189.3, 99.9%=513.6
&#x100084; set B:7900 n=10431, avg=12.85, med=1.19, 99%=206.1, 99.9%=796.8
&#x100084; set C:7900 n=10556, avg=17.29, med=12.97,99%=132.2, 99.9%=322.9
&#x100084; set D:7900 n=10164, avg=7.34,  med=0.64, 99%=98.8, 99.9%=344.4
&#x100084; set E:7900 n=10552, avg=7.18,  med=2.33, 99%=73.6, 99.9%=204.8
&#x100084; set F:7900 n=10337, avg=17.79, med=15.31, 99%=109.0, 99.9%=369.5
```

BeansDB设计实现（非常难得的中文资料）
PPT

Nuclear

人人网研发中的数据库
详见：
http://ugc.renren.com/2010/01/21/ugc-nuclear-guide-use/
http://ugc.renren.com/2010/01/28/ugc-nuclear-guide-theory/

两个设计上的**Tips**

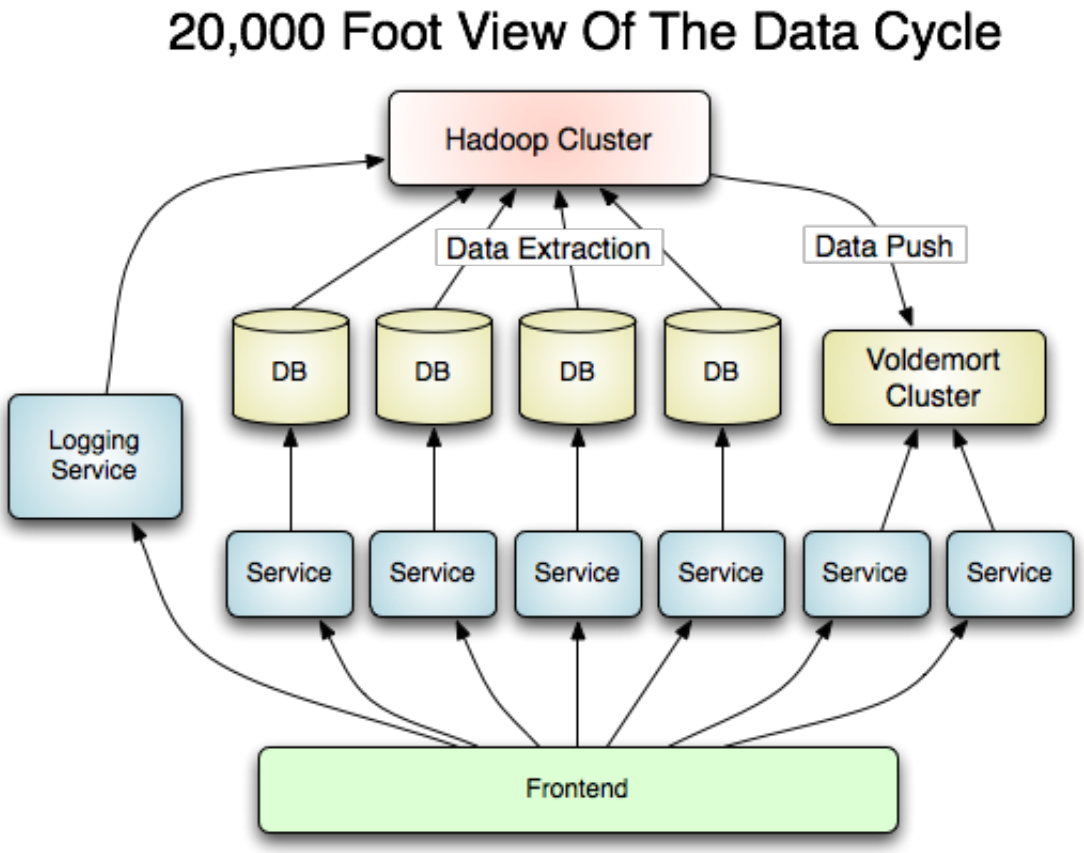
1. 万事皆异步
我们在编码的过程中走了一些弯路，同步的操作在高并发的情况下带来的性能下降是非常恐怖的，于是乎，Nuclear系统中任何的高并发操作都消除了Block。no waiting, no delay。

2. 根据系统负载控制后台线程的资源占用
Nuclear系统中有不少的后台线程默默无闻的做着各种辛苦的工作，但是它们同样会占用系统资源，我们的解决方案是根据系统负载动态调整线程的运行和停止，并达到平衡。

Voldemort

Voldemort: (can you help)

Voldemort是个和Cassandra类似的面向解决scale问题的分布式数据库系统，Cassandra来自于Facebook这个SNS网站，而Voldemort则来自于Linkedin这个SNS网站。说起来SNS网站为我们贡献了n多的NoSQL数据库，例如Cassandar，Voldemort，Tokyo Cabinet，Flare等等。Voldemort的资料不是很多，因此我没有特别仔细去钻研，Voldemort官方给出Voldemort的并发读 写性能也很不错，每秒超过了1.5万次读写。



其实现在很多公司可能都面临着这个抽象架构图中的类似问题。以 Hadoop 作为后端的计算集群，计算得出来的数据如果要反向推到前面去，用什么方式存储更为恰当? 再放到 DB 里面的话，构建索引是麻烦事；放到 Memcached 之类的 Key-Value 分布式系统中，毕竟只是在内存里，数据又容易丢。Voldemort 算是一个不错的改良方案。

值得借鉴的几点：

- 键(Key)结构的设计，有点技巧；
- 架构师熟知硬件结构是有用的。越大的系统越是如此。
- 用好并行。Amdahl 定律以后出现的场合会更多。

详细：

http://www.dbanotes.net/arch/voldemort_key-value.html
http://project-voldemort.com/blog/2009/06/building-a-1-tb-data-cycle-at-linkedin-with-hadoop-and-project-voldemort/

Dynomite

Dynomite: (can you help)

Kai

KAI: Open Source Amazon Dnamo implementation, Misc: slides ,

未分类

Skynet

全新的Ruby MapReduce实现

2004年，Google提出用于分布式数据处理的MapReduce设计模式，同时还提供了第一个C++的实现。现在，一个名为Skynet的Ruby实现已经由Adam Pisoni发布。
Skynet是可适配、可容错的、可自我更新的，而且完全
是分布式的系统，不存在单一的失败节点。

Skynet和Google在设计上有两点重要的区别：
Skynet无法向工作者（Worker）发送原生代码（Raw code），
Skynet利用结对恢复系统，不同的工作者会互相监控以防失败：
如果有一个工作者由于某种原因离开或者放弃了，就会有另一个工作者发现并接管它的任务。Skynet 也没有所谓的"主"管理进程，只有工作者，它们在任何时间都可以充当任何任务的主管理进程。

Skynet的使用和设置都很容易，这也正是MapReduce这个概念的真正优势。Skynet还扩展了ActiveRecord，加入了MapReduce的特性，比如distributed_find。

你要为Starfish编写一些小程序，它们的代码是你将要构建其中的。如果我没有弄错的话，你无法在同一台机器上运行多种类型的MapReduce作业。Skynet是一个更全面的MR系统，可以运行多种类型的多个作业，比如，各种不同的代码。

Skynet也允许失败。工作者会互相关照。如果一个工作者失败了，无法及时完成任务，另一个工作者将会接起这个任务并尝试完成它。Skynet也支持map_data流，也就是说，即使某个数据集非常庞大，甚至无法放在一个数据结构中，Skynet也可以处理。

什 么是map_data流？大多数时候，在你准备启动一个map_reduce作业时，必须提供一个数据的队列，这些数据已经被分离并将被并行处理。如果队 列过大，以至于无法适应于内存怎么办？在这种情况下，你就要不能再用队列，而应该使用枚举（Enumerable）。Skynet知道去对象的调 用:next或者:each方法，然后开始为"每一个（each）"分离出map_task来。通过这样的方式，不会有人再试图同时创建大量的数据结构。

还 有很多特性值得一提，不过最想提醒大家的是，Skynet能够与你现有的应用非常完美地集成到一起，其中自然包括Rails应用。Skynet甚 至还提供了一个ActiveRecord的扩展，你可以在模型中以分布式的形式执行一些任务。在Geni中，我们使用这项功能来运行特别复杂的移植，它通 常涉及到在数百万的模型上执行Ruby代码。
> Model.distributed_find(:all, :conditions => "id > 20").each(:somemethod)在你运行Skynet的时候，它将在每个模型上执行:somemethod，不过是以分布式的方式（这和你 拥有多少个工作者相关）。它在向模型分发任务前不必进行初始化，甚至不必提前获取所有的id。因此它可以操作无限大的数据集。 用户的反馈如何？

Drizzle

Drizzle可 被认为是键/值存储要解决的问题的反向方案。Drizzle诞生于MySQL（6.0）关系数据库的拆分。在过去几个月里，它的开发者已经移走了大量非核 心的功能（包括视图、触发器、已编译语句、存储过程、查询缓冲、ACL以及一些数据类型），其目标是要建立一个更精简、更快的数据库系统。Drizzle 仍能存放关系数据；正如MySQL/Sun的Brian Aker所说那样：“没理由泼洗澡水时连孩子也倒掉”。它的目标就是，针对运行于16核（或以上）系统上的以网络和云为基础的应用，建立一个半关系型数据 库平台。

比较

可扩展性

	Add Machines Live	Multi-DC Support
Cassandra	X	X
HBase	X	
Riak	X	
Scalaris	X	
Voldemort		Some Code Required

数据和查询模型

	Data Model	Query API
Cassandra	Columnfamily	Thrift
CouchDB	Document	map/reduce views
HBase	Columnfamily	Thrift, REST
MongoDB	Document	Cursor
Neo4J	Graph	Graph
Redis	Collection	Collection
Riak	Document	Nested hashes
Scalaris	Key/value	get/put
Tokyo Cabinet	Key/value	get/put
Voldemort	Key/value	get/put

当你需要查询或更新一个值的一部分时，Key/value模型是最简单有效实现。

面向文本数据库是Key/value的下一步，允许内嵌和Key关联的值. 支持查询这些值数据，这比简单的每次返回整个blob类型数据要有效得多。

Neo4J是唯一的存储对象和关系作为数学图论中的节点和边. 对于这些类型数据的查询，他们能够比其他竞争者快1000s

Scalaris是唯一提供跨越多个key的分布式事务。

持久化设计

	Persistence Design
Cassandra	Memtable / SSTable
CouchDB	Append-only B-tree
HBase	Memtable / SSTable on HDFS
MongoDB	B-tree
Neo4J	On-disk linked lists
Redis	In-memory with background snapshots
Riak	?
Scalaris	In-memory only
Tokyo Cabinet	Hash or B-tree
Voldemort	Pluggable (primarily BDB MySQL)

内存数据库是非常快的，(Redis在单个机器上可以完成每秒100,000以上操作）但是数据集超过内存RAM大小就不行. 而且 Durability (服务器当机恢复数据)也是一个问题

Memtables和SSTables缓冲 buffer是在内存中写(“memtable”)， 写之前先追加一个用于durability的日志中.
但有足够多写入以后，这个memtable将被排序然后一次性作为“sstable.”写入磁盘中，这就提供了近似内存性能，因为没有磁盘的查询seeks开销, 同时又避免了纯内存操作的durability问题.(个人点评 其实Java中的Terracotta早就实现这两者结合)
B-Trees提供健壮的索引，但是性能很差，一般和其他缓存结合起来。

应用篇

eBay 架构经验

- 1、 Partition Everything 切分万物
- 2、 Asynchrony Everywhere 处处异步
- 3、 Automate Everything 全部自动
- 4、 Remember Everything Fails 记录失败
- 5、 Embrace Inconsistency 亲不同是谓大同

- 6、 Expect (R)evolution 预言演变
- 7、 Dependencies Matter 重视依赖
- 8、 Be Authoritative 独断专行
- 9、 Never Enough Data
- 10、Custom Infrastructure 自定义基础设施

淘宝架构经验

- 1、适当放弃一致性
- 2、备份和隔离解决稳定性问题
- 3、分割和异步解决性能问题(类似 eBay 的 Asynchrony Everywhere)
- 4、自动化降低人力成本(类似 eBay 的 Automate Everything)
- 5、产品化管理

Flickr架构经验

- 使得机器自动构建 (Teach machines to build themselves)
- 使得机器自监控(Teach machines to watch themselves)
- 使得机器自修复(Teach machines to fix themselves)
- 通过流程减少 MTTR (Reduce MTTR by streamlining)

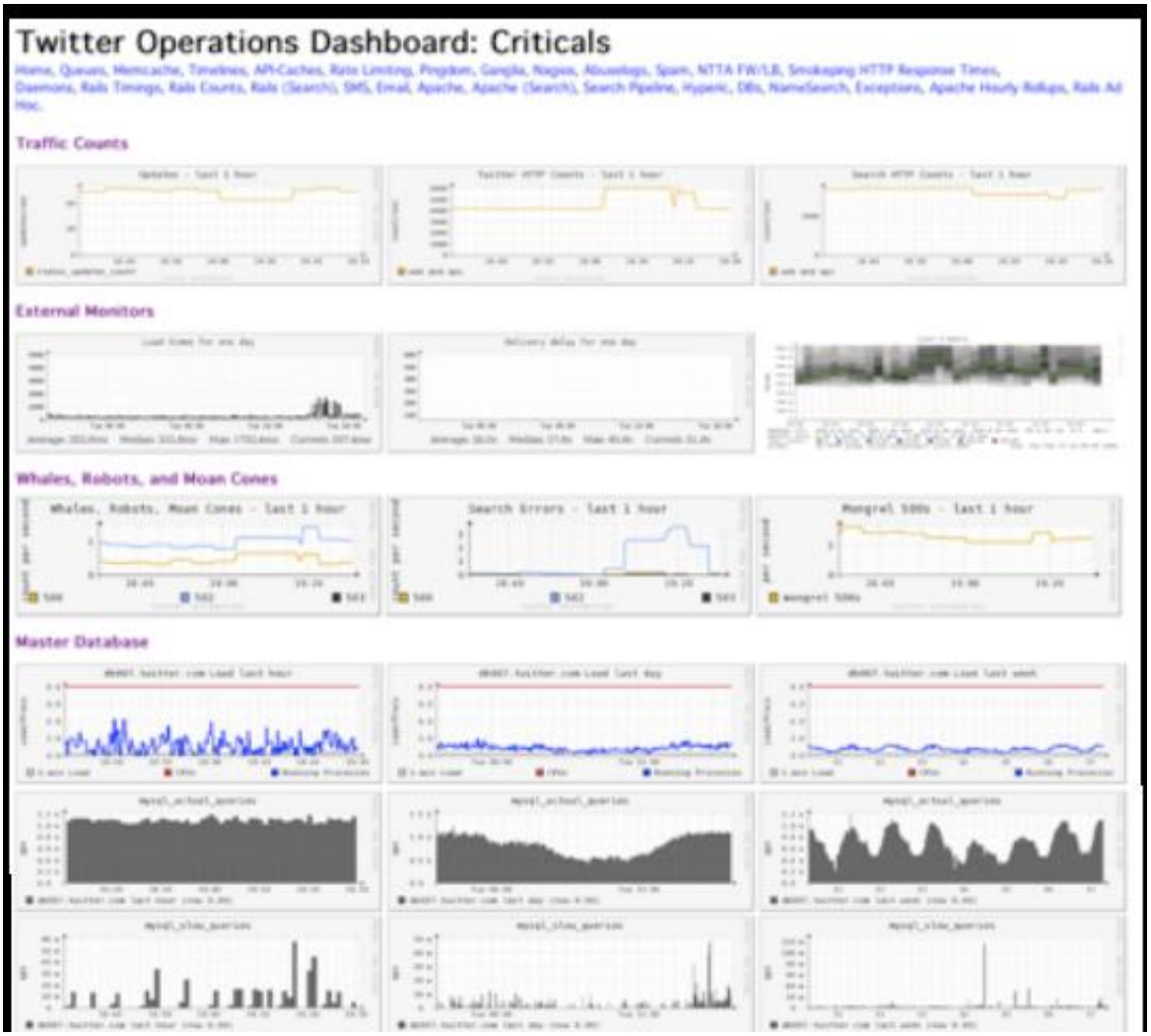
Twitter运维经验

最近看到的另外一个介绍Twitter技术的视频[Slides] [Video (GFWed)]，这是Twitter的John Adams在Velocity 2009的一个演讲，主要介绍了Twitter在系统运维方面一些经验。 本文大部分整理的观点都在Twitter(@xmpp)上发过，这里全部整理出来并补充完整。
Twitter没有自己的硬件，都是由NTTA来提供，同时NTTA负责硬件相关的网络、带宽、负载均衡等业务，Twitter operations team只关注核心的业务，包括**Performance**，**Availability**，**Capacity Planning**容量规划，配置管理等，这个可能跟国内一般的互联网公司有所区别。

运维经验

Metrics

Twitter的监控后台几乎都是图表(critical metrics)，类似驾驶室的转速表，时速表，让操作者可以迅速的了解系统当前的运作状态。联想到我们做的类似监控后台，数据很多，但往往还需要浏览者 做二次分析判断，像这样满屏都是图表的方法做得还不够，可以学习下这方面经验。 据John介绍可以从图表上看到系统的瓶颈-系统最弱的环节(web, mq, cache, db?)
根据图表可以科学的制定系统容量规划，而不是事后救火。



配置管理

每个系统都需要一个自动配置管理系统，越早越好，这条一整理发到Twitter上去之后引起很多回应。

Darkmode

配置界面可以enable/disable 高计算消耗或高I/O的功能，也相当于优雅降级，系统压力过大时取消一些非核心但消耗资源大的功能。

进程管理

Twitter做了一个”Seppaku” patch, 就是将Daemon在完成了n个requests之后主动kill掉，以保持健康的low memory状态，这种做法据了解国内也有不少公司是这样做。

硬件

Twitter将CPU由AMD换成Xeon之后，获得30%性能提升，将CPU由双核/4核换成8核之后，减少了40%的CPU, 不过John也说，这种升级不适合自己购买硬件的公司。

代码协同经验

Review制度

Twitter有上百个模块，如果没有一个好的制度，容易引起代码修改冲突，并把问题带给最终用户。所以Twitter有一强制的source code review制度, 如果提交的代码的svn comment没有”reviewed by xxx”, 则pre-commit脚本会让提交失败, review过的代码提交后会通过自动配置管理系统应用到上百台服务器上。 有@xiaomics同学在Twitter上马上就问，时间成本能否接受？如果有紧急功能怎么办？个人认为紧急修改时有两人在场，一人修改一人

review也不是什么难事。

部署管理

从部署图表可以看到每个发布版本的CPU及latency变化，如果某个新版本latency图表有明显的向上跳跃，则说明该发布版本存在问题。另外在监控首页列出各个模块最后deploy版本的时间，可以清楚的看到代码库的现状。

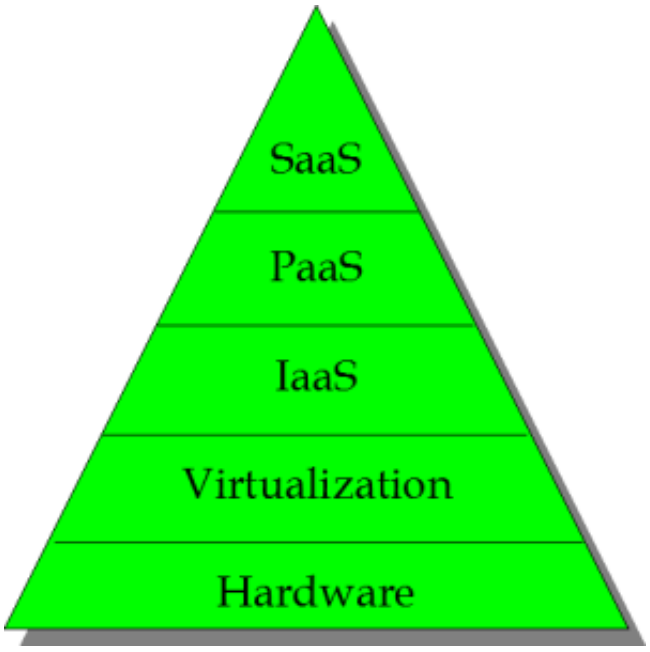
团队沟通

Campfire来协同工作，campfire有点像群，但是更适合协同工作。对于Campfire就不做更多介绍，可参考Campfire官方说明。

Cache

- Memcache key hash, 使用FNV hash 代替 MD5 hash，因为FNV更快。
- 开发了Cache Money plugin(Ruby), 给应用程序提供**read-through, write-through cache**, 就像一个db访问的钩子，当读写数据库的时候会自动更新cache, 避免了繁琐的cache更新代码。
- "Evictions make the cache unreliable for important configuration data"，Twitter使用memcache的一条经验是，不同类型的数据需放在不同的mc,避免eviction，跟作者前文Memcached数据被踢(evictions>0)现象分析中的一些经验一致。
- Memcached SEGVs, Memcached崩溃(cold cache problem)据称会给这种高度依赖Cache的Web 2.0系统带来灾难，不知道Twitter具体怎么解决。
- 在Web层Twitter使用了Varnish作为反向代理，并对其评价较高。

云计算架构



作者认为，金字塔概念最能说明每一层的大小，它也表达了每 个层是依赖前层的消息传递。在概念上，硬件是基础和广泛层。SaaS层是顶峰，也是最轻层。这种观点是来自于将购买SaaS的的最终用户角度。对于一个非 常大的企业内部，PaaS平台层将是顶峰。使用内部开发的软件的内部各部门将实现他们的顶峰SaaS。还要注意：大小和层位置并不一定等同于重要性。硬件 层可能是最重要的，因为它是所有超过一定点的商品。

硬件层**The Hardware Layer**
必须考虑容错和冗余，大部分人认为没有容错硬件廉价商品。冗余和容错处理在软件层内，硬件预计要失败的，当然故障多电源容错服务器，RAID磁盘阵列也是必要的。

虚拟层**The Virtualization Layer**
基于操作系统OS的虚拟化层，虚拟资源能够在线即时增加拓展，允许供应商提供基础设施作为服务(SaaS)，VMware，Citrix公司，Sun都提供虚拟化产品。

The IaaS Layer
提 供和控制的基于虚拟层的计算方式，终端用户能够精确控制每个虚拟机没分钟每小时耗费多少钱。比如提供一个共同的接口，如门户网站暴露的API，允许最终用 户创建和配置虚拟机模板的需求。最终用户还可以控制何时打开或破坏虚拟机，以及如何在虚拟机互相联网。在这个领域的主要竞争者例子是亚马逊网络服务的 EC2，S3和数据库服务。

The PaaS Layer
这一层的目的是尽量减少部署云的复杂性和麻烦，最终用户 利用和开发的这层的API和编程语言。两个很好的例子是谷歌的App Engine 和Force.com平台，在App Engine中，谷歌公开云存储，平台和数据库，以及使用Python和Java编程语言的API。开发人员能够编写应用程序并部署到这一层中，后端可伸缩性架构设计完全交给谷歌负责，最终用户完全不必担心管理基础设施。Force.com平台类似，但采用了自定义的编程语言名为Apex。如果你是一个大型企业寻求内部开发应用的部署，这层是你的顶峰。

The SaaS Layer
如 果您是中小型企业（SME）和大企业不希望开发自己的应用程序时，SaaS的层是你的顶峰（是你将直接面对的）。您只是进行有兴趣地采购如电子邮件或客户 关系管理服务，这些功能服务已经被供应商开发成功，并部署到云环境中了，您只需验证的应用是否符合你的使用需要，帐单可以基于包月租费等各种形式，，作为 最终用户的您不会产生开发和维护拓展应用程序软件的任何成本。越来越多的企业订阅Salesforce.com和Sugar CRM的SaaS产品。

反模式

单点失败（**Single Point of Failure**）

大部分的人都坚持在单一的设备上部署我们的应用，因为这样部署的费用会比较低，但是我们要清楚任何的硬件设备都会有失败的风险的，这种单点失败会严重的影响用户体验甚至是拖垮你的应用，因此除非你的应用能容忍失败带来的损失，否则得话应该尽量的避免单点风险，比如做冗余，热备等。

同步调用

同步调用在任何软件系统中都是不可避免的，但是我们软件工程师必须明白同步调用给软件系统带来的问题。如果我们将应用程序串接起来，那么系统的可用性就会低于任何一个单一组件的可用性。比如组件A同步调用了组件B，组件A的可用性为99.9%，组件B的可用性为99.9%,那么组件A同步调用组件B的可用性就是99.9% * 99.9%=99.8%。同步调用使得系统的可用性受到了所有串接组件可用性的影响，因此我们在系统设计的时候应该清楚哪些地方应该同步调用，在不需要同步调用的时候尽量的进行异步的调用（而我这里所说的异步是一种基于应用的异步，是一种设计上的异步，因为J2EE目前的底层系统出了JMS是异步API以外，其它的API都是同步调用的，所以我们也就不能依赖于底层J2EE平台给我们提供异步性，我们必须从应用和设计的角度引入异步性）

不具备回滚能力

虽然对应用的每个版本进行回滚能力测试是非常耗时和昂贵的，但是我们应该清楚任何的业务操作都有可能失败，那么我们必须为这种失败作好准备，需要对系统的用户负责，这就要求系统一定要具有回滚的能力，当失败的时候能进行及时的回滚。（说到回滚大家可能第一时间想到的是事务的回滚，其实这里的回滚应该是一种更宽泛意义的回滚，比如我们记录每一次的失败的业务操作，这样在出现错误的时候就不是依靠于事务这种技术的手段，而是通过系统本身的回滚能力来进行回滚失败业务操作）。

不记录日志

日志记录对于一个成熟稳定的系统是非常重要的，如果我们不进行日志记录，那么我就很难统计系统的行为。

无切分的数据库

随着系统规模的慢慢变大，我们就需要打破单一数据的限制，需要对其进行切分。

无切分的应用

系统在规模小的时候，也许感觉不出无切分的应用带来的问题，但是在目前互联网高速发展的时代，谁能保证一个小应用在一夜或者是几夜以后还是小应用呢？说不定哪天，我们就发现应用在突如其来的访问量打击的支离破碎。因此我们就需要让我们的系统和我们一样具有生命力，要想让系统具有应付大负载的能力，这就要求我们的应用具有很好的伸缩性，这也就要求应用需要被良好的切分，只有进行了切分，我们才能对单一的部门进行伸缩，如果应用是一块死板的话，我们是没有办法进行伸缩的。就好比火车一样，如果火车设计之初就把他们设计为一体的，那么我们还怎么对火车的车厢进行裁剪？因此一个没有切分的应用是一个没有伸缩性和没有可用性的应用。

将伸缩性依赖于第三方厂商

如果我们的应用系统的伸缩性依赖于第三方的厂商，比如依赖于数据库集群，那么我们就为系统的伸缩性埋下了一个定时炸弹。因为只有我们自己最清楚我们自己的应用，我们应该从应用和设计的角度出发去伸缩我们的应用，而不是依赖于第三方厂商的特性。

OLAP

联机分析处理 (OLAP) 的概念最早是由关系数据库之父E.F.Codd于1993年提出的，他同时提出了关于OLAP的12条准则。OLAP的提出引起了很大的反响，OLAP作为一类产品同联机事务处理 (OLTP) 明显区分开来。

OLAP报表产品最大的难点在哪里？

目前报表工具最大的难点不在于报表的样式（如斜线等），样式虽较繁琐但并非本质困难。最根本的难点在于业务 部门知道报表代表的真正含义，却不知道报表的数据统计模型模型；而IT部门通过理解业务部门的描述，在数据库端进行设置数据统计模型，却对报表本身所代表 的价值很难理解。

说起来有点深奥，其实并不复杂，OLAP最基本的概念只有三个：多维观察、数据钻取、CUBE运算。

关于CUBE运算：OLAP分析所需的原始数据量是非常庞大的。一个分析模型，往往会涉及数百万、数千万条数据，甚至更多；而分析模型中包含多个维数据，这些维又可以由浏览者作任意的提取组合。这样的结果就是大量的实时运算导致时间的延滞。

我们可以设想，一个1000万条记录的分析模型，如果一次提取4个维度进行组合分析，那么实际的运算次数将 达到4的1000次方的数量。这样的运算量将导致数十分钟乃至更长的等待时间。如果用户对维组合次序进行调整，或增加、或减少某些维度的话，又将是一个重 新的计算过程。

从上面的分析中，我们可以得出结论，如果不能解决OLAP运算效率问题的话，OLAP将是一个毫无实用价值的概念。那么，一个成熟产品是如何解决这个问题的呢？这涉及到OLAP中一个非常重要的技术——数据CUBE预运算。

一个OLAP模型中，度量数据和维数据我们应该事先确定，一旦两者确定下来，我们可以对数据进行预先的处理。在正式发布之前，将数据根据维进行最大

限度的聚类运算，运算中会考虑到各种维组合情况，运算结果将生成一个数据CUBE，并保存在服务器上。

这样，当最终用户在调阅这个分析模型的时候，就可以直接使用这个CUBE，在此基础上根据用户的维选择和维组合进行复运算，从而达到实时响应的效果。

NOSQL们背后的共有原则

几个星期之前，我写了一篇文章描述了常被称作 NOSQL 的一类新型数据库的背后驱动。几个星期之前，我在Qcon上发表了一个演讲，其中，我介绍了一个可伸缩（scalable）的 twitter 应用的构建模式，在我们的讨论中，一个显

而易见的问题就是数据库的可扩展性问题。要解答这个问题，我试图寻找隐藏在各种 NOSQL 之后的共有模式，并展示他们是如何解决数据库可扩展性问题的。在本文中，我将尽力勾勒出这些共有的原则。

假设失效是必然发生的

与我们先前通过昂贵硬件之类的手段尽力去避免失效的手段不同，NOSQL实现都建立在硬盘、机器和网络都会失效这些假设之上。我们需要认定，我们不 能彻底阻止这些时效，相反，我们需要让我们的系统能够在即使非常极端的条件下也能应付这些失效。Amazon S3 就是这种设计的一个好例子。你可以在我最近的文章 Why Existing Databases (RAC) are So Breakable! 中找到进一步描述。哪里，我介绍了一些来自 Jason McHugh 的讲演的面向失效的架构设计的内容（Jason 是在 Amazon 做 S3 相关工作的高级工程师）。

对数据进行分区

通过对数据进行分区，我们最小化了失效带来的影响，也将读写操作的负载分布到了不同的机器上。如果一个节点失效了，只有该节点上存储的数据受到影响，而不是全部数据。

保存同一数据的多个副本

大部分 NOSQL 实现都基于数据副本的热备份来保证连续的高可用性。一些实现提供了 API，可以控制副本的复制，也就是说，当你存储一个对象的时候，你可以在对象级指定你希望保存的副本数。在 GigaSpaces，我们还可以立即复制一个新的副本到其他节点，甚至在必要时启动一台新机器。这让我们不比在每个节点上保存太多的数据副本，从而降低 总存储量以节约成本。
你还可以控制副本复制是同步还是异步的，或者两者兼有。这决定了你的集群的一致性、可用性与性能三者。对于同步复制，可以牺牲性能保障一致性和可用 性（写操作之后的任意读操作都可以保证得到相同版本的数据，即使是发生失效也会如此）。而最为常见的 GigaSpaces 的配置是同步副本到被分界点，异步存储到后端存储。

动态伸缩

要掌控不断增长的数据，大部分 NOSQL 实现提供了不停机或完全重新分区的扩展集群的方法。一个已知的处理这个问题的算法称为一致哈希。有很多种不同算法可以实现一致哈希。
一个算法会在节点加入或失效时通知某一分区的邻居。仅有这些节点受到这一变化的影响，而不是整个集群。有一个协议用于掌控需要在原有集群和新节点之间重新分布的数据的变换区间。
另一个（简单很多）的算法使用逻辑分区。在逻辑分区中，分区的数量是固定的，但分区在机器上的分布式动态的。于是，例如有两台机器和1000个逻辑 分区，那么每500个逻辑分区会放在一台机器上。当我们加入了第三台机器的时候，就成了每 333 个分区放在一台机器上了。因为逻辑分区是轻量级的（基于内存中的哈希表），分布这些逻辑分区非常容易。
第二种方法的优势在于它是可预测并且一致的，而使用一致哈希方法，分区之间的重新分布可能并不平稳，当一个新节点加入网络时可能会消耗更长时间。一个用户在这时寻找正在转移的数据会得到一个异常。逻辑分区方法的缺点是可伸缩性受限于逻辑分区的数量。
更进一步的关于这一问题的讨论，建议阅读 Ricky Ho 的文章 NOSQL Patterns 。

查询支持

在这个方面，不同的实现有相当本质的区别。不同实现的一个共性在于哈希表中的 key/value 匹配。一些市县提供了更高级的查询支持，比如面向文档的方法，其中数据以 blob 的方式存储，关联一个键值对属性列表。这种模型是一种无预定义结构的（schema-less）存储，给一个文档增加或删除属性非常容易，无需考虑文档结 构的演进。而 GigaSpaces 支持很多 SQL 操作。如果 SQL查询没有指出特定的简直，那么这个查询就会被并行地 map 到所有的节点去，由客户端完成结果的汇聚。所有这些都是发生在幕后的，用户代码无需关注这些。

使用 **Map/Reduce** 处理汇聚

Map/Reduce 是一个经常被用来进行复杂分析的模型，经常会和 Hadoop 联系在一起。 map/reduce 常常被看作是并行汇聚查询的一个模式。大部分 NOSQL 实现并不提供 map/reduce 的内建支持，需要一个外部的框架来处理这些查询。对于 GigaSpaces 来说，我们在 SQL 查询中隐含了对 map/reduce 的支持，同时也显式地提供了一个称为 executors 的 API 来支持 map/reduce。在质疑模型中，你可以将代码发送到数据所在地地方，并在该节点上直接运行复杂的查询。
这方面的更多细节，建议阅读 Ricky Ho 的文章 Query Processing for NOSQL DB 。

基于磁盘的和内存中的实现

NOSQL 实现分为基于文件的方法和内存中的方法。有些实现提供了混合模型，将内存和磁盘结合使用。两类方法的最主要区别在于每 GB 成本和读写性能。
最近，斯坦福的一项称为"The Case for RAMCloud"的调查，对磁盘和内存两种方法给出了一些性能和成本方面的有趣的比较。总体上说，成本也是性能的一个函数。对于较低性能的实现，磁盘方 案的成本远低于基于内存的方法，而对于高性能需求的场合，内存方案则更加廉价。

> 内存云的显而易见的缺点就是单位容量的高成本和高能耗。对于这些指标，内存云会比纯粹的磁盘系统差50到100 倍，比使用闪存的系统差5-10倍（典型配置情况和指标参见参考文献[1]）。内存云同时还比基于磁盘和闪存的系统需要更多的机房面积。这样，如果一个应 用需要存储大量的廉价数据，不需要高速访问，那么，内存云将不是最佳选择。
> 然而，对于高吞吐量需求的应用，内存云将更有竞争力。当 使用每次操作的成本和能量作为衡量因素的时候，内存云的效率是传统硬盘系统的 **100** 到 **1000** 倍，是闪存系统的 **5-10** 倍。因此，对于高吞吐量需求的系统来说，内存云不仅提供了高性能，也提供了高能源效率。同时，如果使用 **DRAM** 芯片提供的低功耗模式，也可以降低内存云的功耗，特别是在系统空闲的时候。此外，内存云还有一些缺点，一些内存云无法支持需要将数据在 多个数据中心之间进行数据复制。对于这些环境，更新的时延将主要取决于数据中心间数据传输的时间消耗，这就丧失了内存云的时延方面的优势。此外，跨数据中 心的数据复制会让内存云数据一致性更能难保证。不过，内存云仍然可以在夸数据中心的情况下提供低时延的读访问。

仅仅是炒作?

近来我见到的最多的问题就是 "NOSQL 是不是就是炒作？" 或 "NOSQL 会不会取代现在的数据库？"
我的回答是——NOSQL 并非始于今日。很多 NOSQL 实现都已经存在了十多年了，有很多成功案例。我相信有很多原因让它们在如今比以往更受欢迎了。首先是由于社会化网络和云计算的发展，一些原先只有很高端的 组织才会面临的问题，如今已经成为普遍问题了。其次，已有的方法已经被发现无法跟随需求一起扩展了。并且，成本的压力让很多组织需要去寻找更高性价比的方 案，并且研究证实基于普通廉价硬件的分布式存储解决方案甚至比现在的高端数据库更加可靠。（进一步阅读）所有这些导致了对这类"可伸缩性优先数据库"的需求。这里，我引用 AWS团队的接触工程师、VP， James Hamilton 在他的文章 One Size Does Not Fit All 中的一段话：

> "伸缩性优先应用是那些必须具备无限可伸缩性的应用，能够不受限制的扩展比更丰富的功能更加重要。这些应用包括很多需要高 可伸缩性的网站，如 Facebook, MySpace, Gmail, Yahoo 以及 Amazon.com。有些站点实际上使用了关系型数据库，而大部分实际上并未使用。这些服务的共性在于可扩展性比功能公众要，他们无法泡在一个单一的 RDBMS 上。"

总结一下——我认为，现有的 SQL 数据库可能不会很快淡出历史舞台，但同时它们也不能解决世上的所有问题。NOSQL 这个名词现在也变成了 Not Only SQL，这个变化表达了我的观点。

附

本书不求利，只图学术之便。感谢诸位大牛写了那么多的资料，如果您不愿意被引用，学生会重写相应的章节。
引用网志多篇，由于涵盖太广难以一一校队，特此致歉。

感谢

感谢Jdon,dbanotes,infoq和Timyang.您们分享和撰写了那么多有用的资料。

版本志

V0.1版本在2010.2.21发布，提供了本书的主题框架
v0.2版本在2010.2.24发布，因为一些外界原因，提前发布。完善各个示例，勘误，翻译部分内容。
v0.3版本将在3月份或之后发布

引用

http://www.jdon.com/jivejdon/thread/37999
http://queue.acm.org/detail.cfm?id=1413264
http://www.dbanotes.net/arch/five-minute_rule.html
http://www.infoq.com/cn/news/2009/09/Do-Not-Delete-Data
http://www.infoq.com/cn/news/2010/02/ec2-oversubscribed
http://timyang.net/architecture/consistent-hashing-practice
http://en.wikipedia.org/wiki/Gossip_protocol
http://horicky.blogspot.com/2009/11/nosql-patterns.html
http://snarfed.org/space/transactions_across_datacenters_io.html
http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-paxos.pdf
http://en.wikipedia.org/wiki/Distributed_hash_table
http://hi.baidu.com/knuthocean/blog/item/cca1e711221dcfcca6ef3f1d.html
http://zh.wikipedia.org/wiki/MapReduce
http://labs.google.com/papers/mapreduce.html
http://nosql-database.org/
http://www.rackspacecloud.com/blog/2009/11/09/nosql-ecosystem/
http://www.infoq.com/cn/news/2008/02/ruby-mapreduce-skynet
http://s3.amazonaws.com/AllThingsDistributed/sosp/amazon-dynamo-sosp2007.pdf
http://labs.google.com/papers/bigtable.html
http://www.allthingsdistributed.com/2008/12/eventually_consistent.html
http://www.rackspacecloud.com/blog/2009/11/09/nosql-ecosystem/
http://timyang.net/tech/twitter-operations/
http://blog.s135.com/read.php?394
http://www.programmer.com.cn/1760/