很不错的 blog zgl_dm 的空间

介绍Amazon分布式存储引擎Dynamo

标签： 分布式存储 引擎 vector 存储 数据备份 marshalling

2011-03-31 16:22 5779人阅读 评论(3) 收藏 举报

分类： 分布式（8）

目录(?) [+]

Dynamo 是个什么东东呢 ı 公司的一个分布式存储引擎。 那么这个什么引擎又是什么?

首先，假设一个场景，你的网站要存储用户登陆的IP。这个问题怎么解决呢?传统的方法是用数据库。数据库提供了方便的操作接口，复杂 及事物的保证。

好，现在假设大家都很喜欢你的网站，访问的人越来越多。一个数据库已经处理不过来了。于是你安装了3台数据库主机，把用户分成了三类(男人，女人，IT人;总是有某种方法把用户分成数目大致差不多的几个部分吧)。

每次访问的时候，先看用户属于哪一类，然后直接访问存储那类用户数据的数据库。于是处理能力增加了三倍。这个时候你已经实现了一个分布式的存储引 擎，Dynamo 就是一个类似的东西。只是它的可靠性，可用性等方面更好一点而已。下面我们看看那个简单的分布式存储系统有什么不方便的地方，而Dynamo是如何解决 的。

简单分布式系统实现云存储可能存在的问题

先列举一下简单的分布式系统可能存在的问题吧：

1 很难扩容 ：如果现在业务发展迅速，3台主机撑不住了，需要加到5台主机，那要如何处理呢?首先要更改分类方法，把用户分成5类，然后重新迁移已经存在的数据。你要在网站上贴个条子，“系统维护中”，然后开始伟大的迁移工程，等到终于迁移完成，发现其实3台也不用了，用户都走光了。

2 数据可靠性 无法保证：有一天，发现有一台数据库服务器的硬盘坏了，这下麻烦就来了，本来网站就不赚钱，没用什么高档机器，只有一个定期的增量备份而已。经过一天复杂的恢复工作，你还要对部分用户说，麻烦你们把做过的事情再做一遍啊。

3 单点问题 ：负责把用户分类，然后决定使用哪个数据服务器的那台主机是网站的命根子啊，它如果宕机，所有的数据都不能访问了，它如果满负荷了，增加数据服务器也不会对整体性能有帮助。我好像看到一台贴满着驱邪保平安符咒的pc server。

这几个问题，看似不大，解决起来还真的不容易呢。尤其是想到自己的网站也许有一天也会和google有一样多的用户(可能因为你是天才或者google快倒闭了)。现在我们看看 Dynnamo 是怎么解决的吧。

http://hi.baidu.com/beibeiboo/blog/item/71654029e81001f498250a0f.html Dynamo虚节点思想解决扩容问题

Dynamo虚节点思想解决扩容问题，这个问题实际上是数据分布方式的问题(怎么分组)。最简单最容易想到的就是根据资源数目对数据进行哈希分布，比如算 出一个哈希值，然后对资源数取模。这种简单处理的结果就是当资源数变化的时候，每个数据重新取模后，其分布方式都可能变化，从而需要迁移大量的数据。

举个简单的例子来说明一下，假设我的数据是自然数(1-20)，资源现在是三台主机(A，B，C)，采用取模分配方式，那么分配后A主机的数据为 (1，4，7，10，13，16，19)，B为(2，5，8，11，14，17，20) C(3，6，9，12，15，18) 现在增加一台主机D，重新分布后的结果是A(1，5，9，13，17) B(2，6，10，14，18) C(3，7，11，15，19) D(4，8，12，15，20) 。

可以看到，有大量的数据需要从一台主机迁移到另外一台主机。这个迁移过程是很消耗性能的。需要找到一种方式来尽可能减少对现存数据的影响(没有影响当然也不可能，那说明新添加的主机没有数据)。

Dynamo 采用的是 consistent hashing 来解决这个问题的。 那么我们先来了解一下什么是consistent hashing。先想象一个圆，或者你自己的手表表面，把它看成是一个首尾相接的数轴，现在我们的数据，自然数，已经分布到这个圆上了，我们可以把我们的资源采用某种方式，随机的分布到这个圆上(图1-1)。

C++的运算符重载 (81905)
python MySQLdb安装和 (22093)
Nginx 安装与配置 (21759)
linux rename 用法 (21427)
Linux 编译安装Boost (13053)
使用Python的logging.cor (11499)
K-MEANS算法的工作原理 (10168)
Redis之高可用方案 (9181)
如何重置Ubuntu的Unity、 (8205)
atoi 在 Unicode 模式下的 (7620)

评论排行

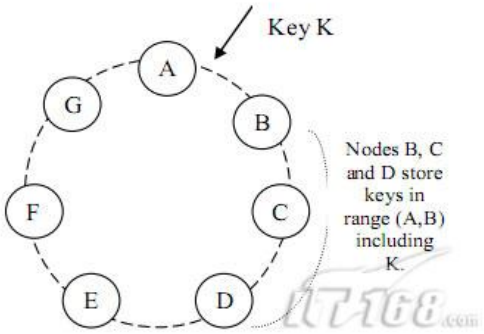
C++的运算符重载 (15)
oracle的表空间、分区表 (5)
K-MEANS算法的工作原理 (5)
Google Protocol Buffers (3)
介绍Amazon分布式存储 (3)
指定下标删除 vector 中的 (2)
Nginx 安装与配置 (2)
Visual C++ ADO数据库编 (2)
Using the Flexgrid in VC (2)
atoi 在 Unicode 模式下的 (2)

推荐文章

* Android-多列表的项目（Rxjava+Rtrofit+Recyclerview+Gli封装）之（一）项目架构
* 为什么Go语言在中国格外的"火"
* Node.js websocket 使用socket.io库实现实时聊天室
* CSDN日报20170219——《程序员的沟通之痛》
* iOS狂暴之路---视图控制器(UIViewController)使用详解

最新评论

Nginx 安装与配置
随心_: https://chenxiaoqiong.github.io/
使用Python的logging.config
哈士奇说喵: 要是在修改下代码格式和文章排版，就更好了，多谢
C++的运算符重载
eternal_an: 感谢^_^
CStdioFile 追加文件内容
yudiandian2014: thank you
Quorum NWR
qq_35550689: 感谢分享，看英文的正在郁闷呢
介绍Amazon分布式存储引擎Dyn
sinat_17076137: 写的真好！
C++的运算符重载
NEET君: 谢谢。
C++的运算符重载
The Scenery Begins: 讲的很好，受教了
secureCRT sftp 使用
bnuchenzhiyan123: 太简单，没有命令全值
C++的运算符重载



【虎.无名：和使用memcached技术类似，在节点变更时提高命中率 。】

现在我们让每一个资源负责它和上一个资源之间的数据，就是说A来负责区间(C，A]，B来负责区间(A，B]，C负责区间(B，C]。采用这种策略，当我 们增加一个资源主机的时候，比如D，那么我们只需要影响新节点相邻的节点A所负责的范围(只需要将A中(C，D]这个区间的数据迁移到D上)就可以了。

因为资源节点是随机分布到数据圆上的，所以当资源节点的数量足够多的时候，可以认为每个节点的负载基本是均衡的。这是原始的consistent hashing 。

Dynamo并没有采用这个模型。这个理想的理论模型跟现实之间有一个问题，在这个理论模型上，每个资源节点的能力是一样的。 我的意思是，他们有相同的cpu，内存，硬盘等，也就是有相同的处理能力。可现实世界，我们使用的资源却各有不同，新买的n核机器和老的奔腾主机一起为了节约成本而合作。如果只是这么简单的把机器直接分布上去，性能高的机器得不到充分利用，性能低的机器处理不过来。

这个问题怎么解决呢?Dynamo 使用的方法是虚节点 。把上面的A B C等都想象成一个逻辑上的节点。一台真实的物理节点可能会包含几个虚节点(逻辑节点)，也可能只包含一个，看机器的性能而定 。

等等，好像我们的网站还没发展成 google 呢，我们能使用的硬件资源还不多，比如就4台主机。这个时候采用上面的方式，把资源随机分布上去，几乎一定会不均衡。这要怎么办呢?我们可以把那个数据圆分成Q等份(每一个等份就是一个虚节点)，这个Q要远大于我们的资源数 。

现在假设我们有S个资源，那么每个资源就承担Q/S个等份。 当一个资源节点离开系统的时候，它所负责的等份要重新均分到其他资源节点上，一个新节点加入的时候，要从其他的节点"偷"到一定数额的等份。

这个策略下，当一个节点离开系统的时候，虽然需要影响到很多节点，但是注意，迁移的数据总量只是离开那个节点的数据量。 同样，一个新节点的加入，迁移的数据总量也只是一个新节点的数据量。 之所以有这个效果是因为Q的存在，使得增加和减少机器的时候不需要对已有的数据做重新哈希 。这个策略的要求是Q>>S (其实还有存储备份的问题，现在还没介绍到，假设每个数据存储N个备份 则要满足Q>>S*N)。如果业务快速发展，使得不断的增加主机，从而导致Q不再满足Q>>S，那么这个策略将不断的退化。

http://hi.baidu.com/beibeiboo/blog/item/c04eed0ad4aac41594ca6b0e.html Dynamo的三点备份模型

第二个问题是数据可靠性，因为我们使用的是廉价的pc机，硬盘损毁或者是其他原因导致的主机不可用是很经常的事情。

做这样一个估算，假设一台pc机平均三年就会有一次失效，不可用。那么当一个一千台机器的集群，基本上每天都有机器坏掉，所以某主机不可用是常态，系统必须可以在这样的情况下继续提供服务 (哦 虽然你的网站现在刚刚只需要4台主机，可是别忘了，它要成长成为google的)。

当然，廉价pc的好处就是便宜。所以我们可以增加系统中数据的备份来使得系统在某台机器挂掉的时候仍旧可用。大家最先想到的方案可能就是对每个节点，建立一个备份节点，如果主节点坏掉了，备份节点可以立刻顶上去(双机热备)。

但是仔细想一下，这个方案是让人不放心的。因为当一主一备中的某一台机器坏掉，另外一台就成了一个单点运行的节点。这个时候另外一个节点一旦发生错误，服务就变得不可用，数据也有可能丢失。在一个要求高可靠性的系统上，这是不可忍受的。

我们刚刚估算了一个大的集群每天都有机器挂掉。而这种错误，一定要人工介入才能解决。想想系统管理员每天在机房里更换主机的情景以及其他不可预料情况(系统管理员休假或者新买的主机没按时到货等)，再想想系统每天都有节点在单点运行，真的是很可怕的事情。

事实上，一般工业界认为比较安全的备份数应该是3份。 好，那么我们看看做这个备份的时候需要注意的问题。首先，如何选择备份节点。我们可以简单的选择顺序上的后两个节点为备份节点，比如存在节点A的数据，备份到节点B和C。但是当我们前面引入了虚节点的概念的时候就要注意了，有可能C节点和A节点在同一台物理机器上，这个时候就不能选择C做为A的备份了 。

下一个问题，当一个节点离开系统的时候，比如宕机，这个节点上存储的信息需要继续备份到其它节点上。虽然节点离开了系统，但是因为备份的存在，我们通过其他节点可以恢复出本节点的所有信息，因为该节点的离开，这部分信息的备份数会比要求的备份数少一，所以需要把这部分数据做再备份 。

同样，当一个节点加入系统，从其他节点偷了数据后，其他节点也需要相应减少备份数。而一个节点如果只是暂时性的不可达，也就是失效一个很短的时间(这种情况是最常发生的)，那么需要其他节点暂时接管这个节点的工作，在其可用的时候，把数据增量传送回该节点 。

http://hi.baidu.com/beibeiboo/blog/item/c3dda333d0713049ac4b5f0d.html 2009-12-02 22:38 NWR模型与同步和异步备份

在设计上述需求的解决方案的时候，还要考虑一个问题，各个节点间数据备份是同步还是异步。 假设我们要求写请求总是尽可能的成功，那么我们的策略是写任何一个节点成功就认为成功。节点之间的数据通过异步形式达成一致，这个时候读请求可能读不到最新写进去的信息。

比如我们一个数据在A B C 三个节点各存一份(系统中有三个备份的时候，下面的讨论都是基于这个假设的)，那么当写A成功后，另外一个进程从节点C读数据，这个时候C还没收到最新的数据，只能给读请求一个较老的版本。这个可能会带来大问题;同样，如果我们希望读请求总能读到正确的数据，那我们的策略是写的时候要等A B C三个节点都写成功了才认为写成功 。这个时候写请求可能要耗较多的时间，甚至根本不能完成(如果有节点不可达)也就是说，系统的一致性，可靠性，原子性，隔离性的问题**(ACID)**是无法同时达到的。只能在其中做出取舍 。

Dynamo 的处理方式是把这个选择权交给用户，这就是它的N W R模型。N代表N个备份，W代表要写入至少W份才认为成功，R表示至少读取R个备份。配置的时候要求W+R > N。 因为W+R > N， 所以 R > N-W 这个是什么意思呢?就是读取的份数一定要比总备份数减去确保写成功的倍数的差值要大 。

也就是说，每次读取，都至少读取到一个最新的版本。从而不会读到一份旧数据。 当我们需要高可写的环境的时候(例如，amazon的购物车的添加请求应该是永远不被拒绝的)我们可以配置W = 1 如果N=3 那么R = 3。 这个时候只要写任何节点成功就认为成功，但是读的时候必须从所有的节点都读出数据。 如果我们要求读的高效率，我们可以配置 W=N R=1。这个时候任何一个节点读成功就认为成功，但是写的时候必须写所有三个节点成功才认为成功 。

大家注意，一个操作的耗时是几个并行操作中最慢一个的耗时。比如R=3的时候，实际上是向三个节点同时发了读请求，要三个节点都返回结果才能认为成功。 假设某个节点的响应很慢，它就会严重拖累一个读操作的响应速度。

http://hi.baidu.com/beibeiboo/blog/item/94b44cd35dd8570a3bf3cf03.html 2009-12-02 22:33 vector clock算法保证版本信息

解决数据版本问题

这里我们需要讨论一下数据版本问题，这个问题不仅仅存在于分布式系统，只是分布式系统的一些要求使得这个问题更复杂。 先 看个简单的例子，用户x对key1做了一次写入操作，我们设值是数字3。然后用户y读取了key1，这个时候用户y知道的值是3。然后用户x对值做了一 个+1操作，将新值写入，现在key1的值是4了。而用户y也做了一次+1操作，然后写入，因为用户y读到的值是3，y不知道这个值现在已经变化了，结果 按照语义本应该是5的值，现在还是4。

解决这个问题常用的方法是设置一个版本值。用户x第一次写入key1 值3的时候，产生一个版本设为v1。用户y读取的信息中包括版本编号v1。当x做了加1把值4写入的时候，告诉server自己拿到的是版本v1，要在 v1的基础上把值改成4。server发现自己保存的版本的确是v1所以就同意这个写入，并且把版本改成v2。 这个时候y也要写入4，并且宣称自己是在版本v1上做的修改。

但是因为server发现自己手里已经是版本v2了，所以server就拒绝y的写入请求，告诉y，版本错误。 这个算法在版本冲突的时候经常被使用。但是刚才我们描述的分布式系统不能简单采用这个方式来实现。

假设我们设置了N=3 W=1。现在x写入key1 值3，这个请求被节点A处理，生成了v1版本的数据。然后x用户又在版本v1上进行了一次key1值4的写操作，这个请求这次是节点C处理的。但是节点C 还没有收到上一个A接收的版本(数据备份是异步进行的)如果按照上面的算法，他应该拒绝这个请求，因为他不了解版本v1的信息。但是实际上是不可以拒绝的，因为如果C拒绝了写请求，实际上W=1这个配置，这个服务器向客户做出的承诺将被打破，从而使得系统的行为退化成W=N的形式。 那么C接收了这个请求，就可能产生前面提到的不一致性。如何解决这个问题呢?

Dynamo 的方法是保留所有这些版本，用vector clock记录版本信息。当读取操作发生的时候返回多个版本，由客户端的业务层来解决这个冲突合并各个版本。当然客户端也可以选择最简单的策略，就是最近一次的写覆盖以前的写 。

vector clock算法保证版本信息

这里又引入了一个vector clock算法，这里简单介绍一下。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 。

解决单点故障问题

最后还有一个单点故障的问题，这个问题的解决要求系统构建的时候是完全分布式，不存在一个核心的 。 例如传统的存储系统里面往往存在一个中心节点，这个单点的存在使得系统在这一点上变得很脆弱。解决方法就是让系统的每个节点都可以承担起所有需要的功能。 这个问题的解决涉及到事情比较多，以亚马逊平台的Dynamo分布式存储引擎来说，有Seed节点的概念，不过本文我们暂时不做过多剖析。

http://storage.it168.com/a2009/1013/757/000000757579_3.shtml 详细解析Dynamo存储引擎

NWR模型与同步和异步备份

在设计上述需求的解决方案的时候，还要考虑一个问题，各个节点间数据备份是同步还是异步。假设我们要求写请求总是尽可能的成功，那么我们的策略是写任何一个节点成功就认为成功。节点之间的数据通过异步形式达成一致，这个时候读请求可能读不到最新写进去的信息。

比如我们一个数据在A B C 三个节点各存一份(系统中有三个备份的时候，下面的讨论都是基于这个假设的)，那么当写A成功后，另外一个进程从节点C读数据，这个时候C还没收到最新的 数据，只能给读请求一个较老的版本。这个可能会带来大问题;同样，如果我们希望读请求总能读到正确的数据，那我们的策略是写的时候要等A B C三个节点都写成功了才认为写成功。这个时候写请求可能要耗较多的时间，甚至根本不能完成(如果有节点不可达)也就是说，系统的一致性，可靠性，原子性，隔离性的问题(ACID)是无法同时达到的。只能在其中做出取舍 。

Dynamo 的处理方式是把这个选择权交给用户，这就是它的N W R模型。N代表N个备份，W代表要写入至少W份才认为成功，R表示至少读取R个备份。配置的时候要求W+R > N 。 因为W+R > N， 所以 R > N-W 这个是什么意思呢?就是读取的份数一定要比总备份数减去确保写成功的倍数的差值要大。

也就是说，每次读取，都至少读取到一个最新的版本。从而不会读到一份旧数据。当我们需要高可写的环境的时候(例如，amazon的购物车的添加请求应该是 永远不被拒绝的)我们可以配置W = 1 如果N=3 那么R = 3。 这个时候只要写任何节点成功就认为成功，但是读的时候必须从所有的节点都读出数据。如果我们要求读的高效率，我们可以配置 W=N R=1。这个时候任何一个节点读成功就认为成功，但是写的时候必须写所有三个节点成功才认为成功。

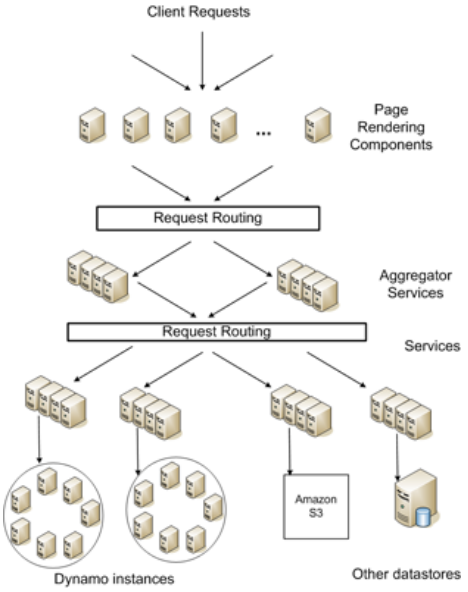
大家注意，一个操作的耗时是几个并行操作中最慢一个的耗时。比如R=3的时候，实际上是向三个节点同时发了读请求，要三个节点都返回结果才能认为成功。假设某个节点的响应很慢，它就会严重拖累一个读操作的响应速度。

http://www.dbanotes.net/techmemo/amazon_dynamo.html Amazon 的 Dynamo 架构

作者: Fenng | 可以转载, 但必须以超链接形式标明文章原始出处和作者信息及版权声明.

http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html Amazon Dynamo 这个高可用、可扩展存储体系 支撑了Amazon 不少核心服务.

先看一个示意图:



从上图可以看出，Amazon 的架构是完全的分布式，去中心化。存储层也做到了分布式 。

Dynamo 概述

Dynamo 的可扩展性和可用性采用的都比较成熟的技术，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。 Dynamo 是完全去中心化的系统，人工管理工作很小。

强调一下 Dynamo 的"额外"特点：
1) 总是可写
2) 可以根据应用类型优化

关键词

Key : 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.
节点**(node)** ：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)
实例**(instance)** ；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。

上面提到的本地持久引擎支持不同的存储引擎 。Dynamo 上最主要的引擎是 Berkeley Database Transactional Data Store(存储处理数百K的对象更为适合），其他还有 BDB **Java** Edition、**MySQL** 以及 一致性内存 Cache 等等 。

三个关键参数 **(N,R,W)**

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 Dynamo 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。

(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。

–待续– 更多阅读：Dynamo学习 — http://donghao.org/2008/10/dynamoni.html

http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html 论文：网页版

4.System Architecture

The architecture of a storage system that needs to operate in a production setting is complex. In addition to the actual data persistence component, the system needs to have scalable and robust solutions for load balancing, membership and failure detection, failure recovery, replica synchronization, overload handling, state transfer, concurrency and job scheduling, request marshalling, request routing, system monitoring and alarming, and configuration management. Describing the details of each of the solutions is not possible, so this paper focuses on the core distributed systems techniques used in Dynamo: partitioning, replication, versioning, membership, failure handling and scaling. Table 1 presents a summary of the list of techniques Dynamo uses and their respective advantages.

Table 1: Summary of techniques used in Dynamo and their advantages.

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

上一篇 Why Vector Clocks are Easy
下一篇 Dynamo虚节点思想解决扩容问题

我的同类文章

分布式（8）

- Hbase伪分布，建表报错 2013-10-27 阅读 581
- 关于Nginx 简单的负载均衡… 2011-04-07 阅读 589
- Dynamo虚节点思想解决扩… 2011-03-31 阅读 792
- Why Vector Clocks are Easy 2011-03-31 阅读 1009
- Brewer's CAP Theorem 2011-03-31 阅读 761
- Quorum NWR 2011-03-31 阅读 2037
- CAP原理与最终一致性 2011-03-31 阅读 688
- 一致性 hash 算法（ consist… 2011-03-31 阅读 648

参考知识库

MySQL知识库
19547 关注 | 1446 收录

Java 知识库
22977 关注 | 1441 收录

.NET知识库
3104 关注 | 827 收录

Java SE知识库
22367 关注 | 468 收录

猜你在找

Ceph—分布式存储系统的另一个选择
全网服务器数据备份解决方案案例实践
Oracle高级管理
4.7.存储类&作用域&生命周期&链接属性-C语言高级专题
Oracle数据库
详细解析Dynamo存储引擎
分布式系统工程实现Bigtable Merge-Dump存储引擎
分布式系统工程实践随机访问KV存储引擎
淘宝分布式 keyvalue 存储引擎Tair安装部署过程及
淘宝分布式 keyvalue 存储引擎Tair安装部署过程及

查看评论

3楼 sinat_17076137 2016-05-02 21:59发表

写的真好！

2楼 Casualet 2015-11-23 23:11发表

写得真好！

1楼 zhuojiajin 2015-01-23 15:59发表

复杂

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC
coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr
Angular Cloud Foundry Redis Scala Django Bootstrap