系统的可扩展性是推动NoSQL运动发展的的主要理由，包含了分布式系统协调，故障转移，资源管理和许多其他特性。这么讲使得NoSQL听起来像是一个大筐，什么都能塞进去。尽管NoSQL运动并没有给分布式数据处理带来根本性的技术变革，但是依然引发了铺天盖地的关于各种协议和算法的研究以及实践。正是通过这些尝试逐渐总结出了一些行之有效的数据库构建方法。在这篇文章里，我将针对NoSQL数据库的分布式特点进行一些系统化的描述。

接下来我们将研究一些分布式策略，比如故障检测中的复制，这些策略用黑体字标出，被分为三段：

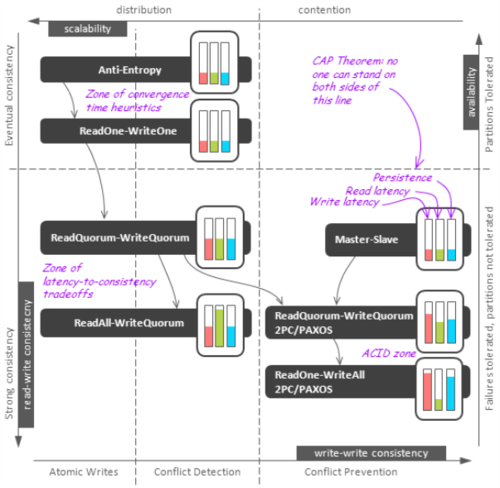
* **数据一致性**。NoSQL需要在分布式系统的一致性，容错性和性能，低延迟及高可用之间作出权衡，一般来说，数据一致性是一个必选项，所以这一节主要是关于**数据复制**和**数据恢复**。
* **数据放置**。一个数据库产品应该能够应对不同的数据分布，集群拓扑和硬件配置。在这一节我们将讨论如何分布以及调整数据分布才能够能够及时解决故障，提供持久化保证，高效查询和保证集群中的资源（如内存和硬盘空间）得到均衡使用。
* **对等系统**。像 leader election 这样的的技术已经被用于多个数据库产品以实现容错和数据强一致性。然而，即使是分散的的数据库（无中心）也要跟踪它们的全局状态，检测故障和拓扑变化。这一节将介绍几种使系统保持一致状态的技术。

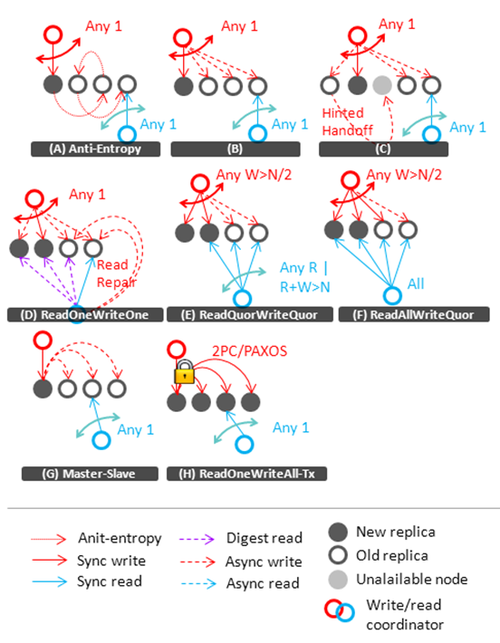
**数据一致性**

众所周知，分布式系统经常会遇到网络隔离或是延迟的情况，在这种情况下隔离的部分是不可用的，因此要保持高可用性而不牺牲一致性是不可能的。这一事实通常被称作“CAP理论”。然而，一致性在分布式系统中是一个非常昂贵的东西，所以经常需要在这上面做一些让步，不只是针对可用性，还有多种权衡。为了研究这些权衡，我们注意到分布式系统的一致性问题是由数据隔离和复制引起的，所以我们将从研究复制的特点开始：

* 可用性。在网络隔离的情况下剩余部分仍然可以应对读写请求。
* 读写延迟。读写请求能够在短时间内处理。
* 读写延展性。读写的压力可由多个节点均衡分担。
* 容错性。对于读写请求的处理不依赖于任何一个特定节点。
* 数据持久性。特定条件下的节点故障不会造成数据丢失。
* **一致性。**一致性比前面几个特性都要复杂得多，我们需要详细讨论一下几种不同的观点。 但是我们不会涉及过多的一致性理论和并发模型，因为这已经超出了本文的范畴，我只会使用一些简单特点构成的精简体系。
* **读写一致性。**从读写的观点来看，数据库的基本目标是使副本趋同的时间尽可能短（即更新传递到所有副本的时间），保证最终一致性。除了这个较弱的保证，还有一些更强的一致性特点：
* **写后读一致性。**在数据项X上写操作的效果总是能够被后续的X上的读操作看见。
* **读后读一致性。**在一次对数据项X的读操作之后，后续对X的读操作应该返回与第一次的返回值相同或是更加新的值。
* **写一致性。**分区的数据库经常会发生写冲突。数据库应当能处理这种冲突并保证多个写请求不会被不同的分区所处理。这方面数据库提供了几种不同的一致性模型：
* **原子写。**假如数据库提供了API，一次写操作只能是一个单独的原子性的赋值，避免写冲突的办法是找出每个数据的“最新版本”。这使得所有的节点都能够在更新结束时获得同一版本，而与更新的顺序无关，网络故障和延迟经常造成各节点更新顺序不一致。 数据版本可以用时间戳或是用户指定的值来表示。Cassandra用的就是这种方法。
* **原子化的读-改-写。**应用有时候需要进行 读-改-写 序列操作而非单独的原子写操作。假如有两个客户端读取了同一版本的数据，修改并且把修改后的数据写回，按照原子写模型，时间上比较靠后的那一次更新将会覆盖前一次。这种行为在某些情况下是不正确的（例如，两个客户端往同一个列表值中添加新值）。数据库提供了至少两种解决方法：
* **冲突预防。** 读-改-写 可以被认为是一种特殊情况下的事务，所以分布式锁或是 PAXOS这样的一致协议都可以解决这种问题。这种技术支持原子读改写语义和任意隔离级别的事务。另一种方法是避免分布式的并发写操作，将对特定数据项的所有写操作路由到单个节点上（可以是全局主节点或者分区主节点）。为了避免冲突，数据库必须牺牲网络隔离情况下的可用性。这种方法常用于许多提供强一致性保证的系统（例如大多数关系数据库，HBase，MongoDB）。
* **冲突检测。**数据库跟踪并发更新的冲突，并选择回滚其中之一或是维持两个版本交由客户端解决。并发更新通常用向量时钟 （这是一种乐观锁）来跟踪，或者维护一个完整的版本历史。这个方法用于 Riak, Voldemort, CouchDB.

现在让我们仔细看看常用的复制技术，并按照描述的特点给他们分一下类。第一幅图描绘了不同技术之间的逻辑关系和不同技术在系统的一致性、扩展性、可用性、延迟性之间的权衡坐标。 第二张图详细描绘了每个技术。





复本因子是4。读写协调者可以是一个外部客户端或是一个内部代理节点。

**我们会依据一致性从弱到强把所有的技术过一遍：**

**（A, 反熵） 一致性最弱，基于策略如下。**写操作的时候选择任意一个节点更新，在读的时候如果新数据还没有通过后台的反熵协议传递到读的那个节点，那么读到的仍然是旧数据。（下一节会详细介绍反熵协议）。这种方法的主要特点是：

* 过高的传播延迟使它在数据同步方面不太好用，所以比较典型的用法是只作为辅助性的功能来检测和修复计划外的不一致。Cassandra就使用了反熵算法来在各节点之间传递数据库拓扑和其他一些元数据信息。
* 一致性保证较弱：即使在没有发生故障的情况下，也会出现写冲突与读写不一致。
* 在网络隔离下的高可用和健壮性。用异步的批处理替代了逐个更新，这使得性能表现优异。
* 持久性保障较弱因为新的数据最初只有单个副本。

**（B） 对上面模式的一个改进是在任意一个节点收到更新数据请求的同时异步的发送更新给所有可用节点。这也被认为是定向的反熵。**

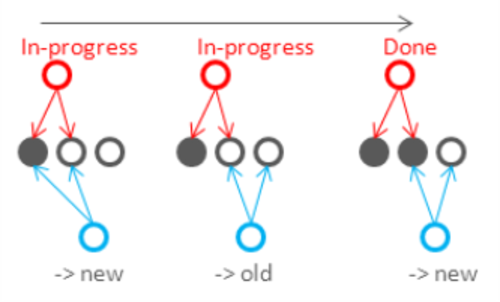
* 与纯粹的反熵相比，这种做法只用一点小小的性能牺牲就极大地提高了一致性。然而，正式一致性和持久性保持不变。
* 假如某些节点因为网络故障或是节点失效在当时是不可用的，更新最终也会通过反熵传播过程来传递到该节点。

**（C） 在前一个模式中，使用提示移交技术可以更好地处理某个节点的操作失败。**对于失效节点的预期更新被记录在额外的代理节点上，并且标明一旦特点节点可用就要将更新传递给该节点。这样做提高了一致性，降低了复制收敛时间。

**（D, 一次性读写）因为提示移交的责任节点也有可能在将更新传递出去之前就已经失效，在这种情况下就有必要通过所谓的读修复来保证一致性。**每个读操作都会启动一个异步过程，向存储这条数据的所有节点请求一份数据摘要（像签名或者hash），如果发现各节点返回的摘要不一致则统一各节点上的数据版本。我们用一次性读写来命名组合了A、B、C、D的技术- 他们都没有提供严格的一致性保证，但是作为一个自备的方法已经可以用于实践了。

**（E, 读若干写若干） 上面的策略是降低了复制收敛时间的启发式增强。**为了保证更强的一致性，必须牺牲可用性来保证一定的读写重叠。 通常的做法是同时写入W个副本而不是一个，读的时候也要读R个副本。

* 首先，可以配置写副本数W>1。
* 其次，因为R+W>N，写入的节点和读取的节点之间必然会有重叠，所以读取的多个数据副本里至少会有一个是比较新的数据（上面的图中 W=2, R=3, N=4 ）。这样在读写请求依序进行的时候（写执行完再读）能够保证一致性（对于单个用户的读写一致性），但是不能保障全局的读一致性。用下面图示里的例子来看，R=2，W=2，N=3，因为写操作对于两个副本的更新是非事务的，在更新没有完成的时候读就可能读到两个都是旧值或者一新一旧：



* 对于某种读延迟的要求，设置R和W的不同值可以调整写延迟与持久性，反之亦然。
* 如果W<=N/2，并发的多个写入会写到不同的若干节点（如，写操作A写前N/2个，B写后N/2个）。 设置 W>N/2 可以保证在符合回滚模型的原子读改写时及时检测到冲突。
* 严格来讲，这种模式虽然可以容忍个别节点的失效， 但是对于网络隔离的容错性并不好。在实践中，常使用”近似数量通过“这样的方法，通过牺牲一致性来提高某些情景下的可用性。

**（F, 读全部写若干）读一致性问题可以通过在读数据的时候访问所有副本（读数据或者检查摘要）来减轻。**这确保了只要有至少一个节点上的数据更新新的数据就能被读取者看到。但是在网络隔离的情况下这种保证就不能起到作用了。

**（G, 主从） 这种技术常被用来提供原子写或者 冲突检测持久级别的读改写。为了实现冲突预防级别，必须要用一种集中管理方式或者是锁。**最简单的策略是用主从异步复制。对于特定数据项的写操作全部被路由到一个中心节点，并在上面顺序执行。这种情况下主节点会成为瓶颈，所以必须要将数据划分成一个个独立的片区（不同片有不同的master），这样才能提供扩展性。

**（H, Transactional Read Quorum Write Quorum and Read One Write All）  更新多个副本的方法可以通过使用事务控制技术来避免写冲突。** 众所周知的方法是使用两阶段提交协议。但两阶段提交并不是完全可靠的，因为协调者失效可能会造成资源阻塞。 PAXOS提交协议是更可靠的选择，但会损失一点性能。 在这个基础上再向前一小步就是读一个副本写所有副本，这种方法把所有副本的更新放在一个事务中，它提供了强容错一致性但会损失掉一些性能和可用性。

**上面分析中的一些权衡有必要再强调一下**

* 一致性与可用性。 严密的权衡已经由CAP理论给出了。在网络隔离的情况下，数据库要么将数据集中，要么既要接受数据丢失的风险。
* 一致性与扩展性。 看得出即使读写一致性保证降低了副本集的扩展性，只有在原子写模型中才可以以一种相对可扩展的方式处理写冲突。原子读改写模型通过给数据加上临时性的全局锁来避免冲突。这表明， 数据或操作之间的依赖，即使是很小范围内或很短时间的，也会损害扩展性。所以精心设计数据模型，将数据分片分开存放对于扩展性非常重要。
* 一致性与延迟。 如上所述，当数据库需要提供强一致性或者持久性的时候应该偏向于读写所有副本技术。但是很明显一致性与请求延迟成反比，所以使用若干副本技术会是比较中允的办法。
* 故障转移与一致性/扩展性/延迟。有趣的是容错性与一致性、扩展性、延迟的取舍冲突并不剧烈。通过合理的放弃一些性能与一致性，集群可以容忍多达 up to 的节点失效。这种折中在两阶段提交与 PAXOS 协议的区别里体现得很明显。这种折中的另一个例子是增加特定的一致性保障，比如使用严格会话进程的“读己所写”，但这又增加了故障转移的复杂性。

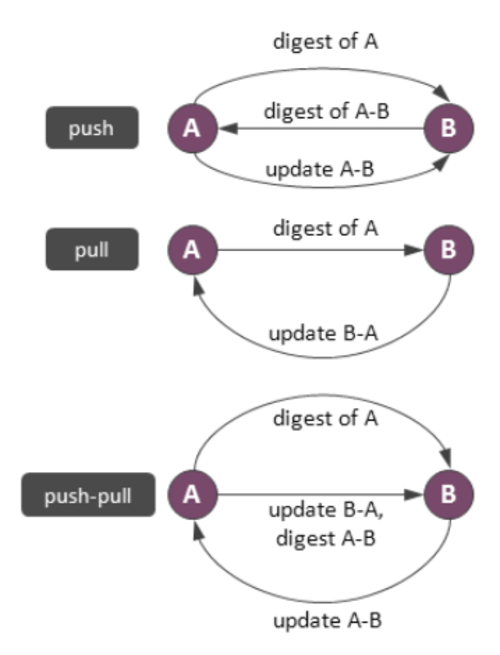
**反熵协议， 谣言传播算法**

让我们从以下场景开始：

有许多节点，每条数据会在其中的若干的节点上面存有副本。每个节点都可以单独处理更新请求，每个节点定期和其他节点同步状态，如此一段时间之后所有的副本都会趋向一致。同步过程是怎样进行的？同步何时开始？怎样选择同步的对象？怎么交换数据？我们假定两个节点总是用较新版本的数据覆盖旧的数据或者两个版本都保留以待应用层处理。

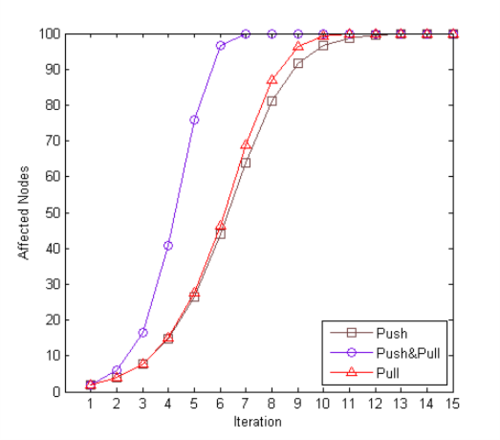
这个问题常见于数据一致性维护和集群状态同步（如集群成员信息传播）等场景。虽然引入一个监控数据库并制定同步计划的协调者可以解决这个问题，但是去中心化的数据库能够提供更好的容错性。去中心化的主要做法是利用精心设计的传染协议，这种协议相对简单，但是提供了很好的收敛时间，而且能够容忍任何节点的失效和网络隔离。尽管有许多类型的传染算法，我们只关注反熵协议，因为NoSQL数据库都在使用它。

反熵协议假定同步会按照一个固定进度表执行，每个节点定期随机或是按照某种规则选择另外一个节点交换数据，消除差异。有三种反风格的反熵协议：推，拉和混合。推协议的原理是简单选取一个随机节点然后把数据状态发送过去。在真实应用中将全部数据都推送出去显然是愚蠢的，所以节点一般按照下图所示的方式工作。



节点A作为同步发起者准备好一份数据摘要，里面包含了A上数据的指纹。节点B接收到摘要之后将摘要中的数据与本地数据进行比较，并将数据差异做成一份摘要返回给A。最后，A发送一个更新给B，B再更新数据。拉方式和混合方式的协议与此类似，就如上图所示的。

反熵协议提供了足够好的收敛时间和扩展性。下图展示了一个在100个节点的集群中传播一个更新的模拟结果。在每次迭代中，每个节点只与一个随机选取的对等节点发生联系。



可以看到，拉方式的收敛性比推方式更好，这可以从理论上得到证明。而且推方式还存在一个“收敛尾巴”的问题。在多次迭代之后，尽管几乎遍历到了所有的节点，但还是有很少的一部分没受到影响。与单纯的推和拉方式相比， 混合方式的效率更高，所以实际应用中通常使用这种方式。反熵是可扩展的，因为平均转换时间以集群规模的对数函数形式增长。

尽管这些技术看起来很简单，仍然有许多研究关注于不同约束条件下反熵协议的性能表现。其中之一通过一种更有效的结构使用网络拓扑来取代随机选取 。在网络带宽有限的条件下调整传输率或使用先进的规则来选取要同步的数据 。摘要计算也面临挑战，数据库会维护一份最近更新的日志以有助于摘要计算。

**最终一致数据类型Eventually Consistent Data Types**

在上一节我们假定两个节点总是合并他们的数据版本。但要解决更新冲突并不容易，让所有副本都最终达到一个语义上正确的值出乎意料的难。一个众所周知的例子是Amazon Dynamo数据库中已经删除的条目可以重现。

我们假设一个例子来说明这个问题：数据库维护一个逻辑上的全局计数器，每个节点可以增加或者减少计数。虽然每个节点可以在本地维护一个自己的值，但这些本地计数却不能通过简单的加减来合并。假设这样一个例子：有三个节点A、B和C，每个节点执行了一次加操作。如果A从B获得一个值，并且加到本地副本上，然后C从B获得值，然后C再从A获得值，那么C最后的值是4，而这是错误的。解决这个问题的方法是用一个类似于向量时钟的数据结构为每个节点维护一对计数器：

**[js]** [view plaincopy](http://www.csdn.net/article/2015-01-30/2823782)[在CODE上查看代码片](https://code.csdn.net/snippets/593506)

1. 1 **class** Counter {
2. 2     **int**[] plus
3. 3     **int**[] minus
4. 4     **int** NODE\_ID
5. 5
6. 6     increment() {
7. 7         plus[NODE\_ID]++
8. 8     }
9. 9
10. 10    decrement() {
11. 11        minus[NODE\_ID]++
12. 12    }
13. 13
14. 14    get() {
15. 15        **return** sum(plus) – sum(minus)
16. 16    }
17. 17
18. 18    merge(Counter other) {
19. 19        **for** i **in** 1..MAX\_ID {
20. 20            plus[i] = max(plus[i], other.plus[i])
21. 21            minus[i] = max(minus[i], other.minus[i])
22. 22        }
23. 23    }
24. 24 }

Cassandra用类似的方法计数。利用基于状态的或是基于操作的复制理论也可以设计出更复杂的最终一致的数据结构。例如，中就提及了一系列这样的数据结构，包括：

* 计数器（加减操作）
* 集合（添加和移除操作）
* 图（增加边或顶点，移除边或顶点）
* 列表（插入某位置或者移除某位置）

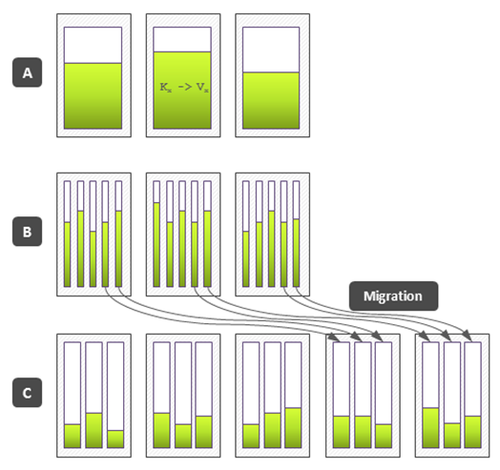
最终一致数据类型的功能通常是有限的，还会带来额外的性能开销。

**数据放置**

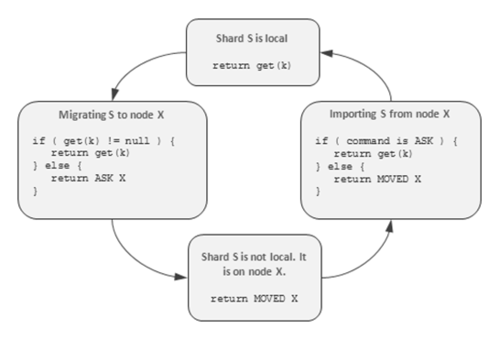
这部分主要关注控制在分布式数据库中放置数据的算法。这些算法负责把数据项映射到合适的物理节点上，在节点间迁移数据以及像内存这样的资源的全局调配。

**均衡数据**

我们还是从一个简单的协议开始，它可以提供集群节点间无缝的数据迁移。这常发生于像集群扩容（加入新节点），故障转移（一些节点宕机）或是均衡数据（数据在节点间的分布不均衡）这样的场景。如下图A中所描绘的场景 – 有三个节点，数据随便分布在三个节点上（假设数据都是key-value型）。



如果数据库不支持数据内部均衡，就要在每个节点上发布数据库实例，如上面图B所示。这需要手动进行集群扩展，停掉要迁移的数据库实例，把它转移到新节点上，再在新节点上启动，如图C所示。尽管数据库能够监控到每一条记录，包括MongoDB, Oracle Coherence, 和还在开发中的 Redis Cluster 在内的许多系统仍然使用的是自动均衡技术。也即，将数据分片并把每个数据分片作为迁移的最小单位，这是基于效率的考虑。很明显分片数会比节点数多，数据分片可以在各节点间平均分布。按照一种简单的协议即可实现无缝数据迁移，这个协议可以在迁移数据分片的时候重定向客户的数据迁出节点和迁入节点。下图描绘了一个Redis Cluster中实现的get（key）逻辑的状态机。

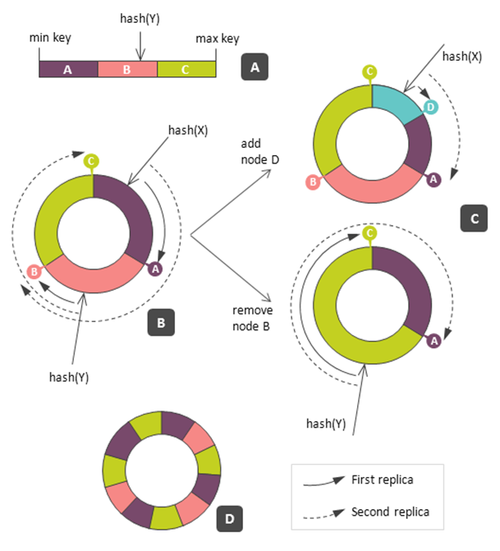


假定每个节点都知道集群拓扑，能够把任意key映射到相应的数据分片，把数据分片映射到节点。如果节点判断被请求的key属于本地分片，就会在本地查找（上图中上面的方框）。假如节点判断请求的key属于另一个节点X，他会发送一个永久重定向命令给客户端（上图中下方的方框）。永久重定向意味着客户端可以缓存分片和节点间的映射关系。如果分片迁移正在进行，迁出节点和迁入节点会标记相应的分片并且将分片的数据加锁逐条加锁然后开始移动。迁出节点首先会在本地查找key，如果没有找到，重定向客户端到迁入节点，假如key已经迁移完毕的话。这种重定向是一次性的，并且不能被缓存。迁入节点在本地处理重定向，但定期查询在迁移还没完成前被永久重定向。

**动态环境中的数据分片和复制**

我们关注的另一个问题是怎么把记录映射到物理节点。比较直接的方法是用一张表来记录每个范围的key与节点的映射关系，一个范围的key对应到一个节点，或者用key的hash值与节点数取模得到的值作为节点ID。但是hash取模的方法在集群发生更改的情况下就不是很好用，因为增加或者减少节点都会引起集群内的数据彻底重排。导致很难进行复制和故障恢复。

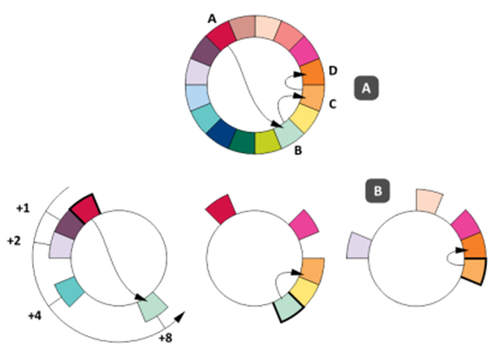
有许多方法在复制和故障恢复的角度进行了增强。最著名的就是一致性hash。网上已经有很多关于一致性hash的介绍了，所以在这里我只提供一个基本介绍，仅仅为了文章内容的完整性。下图描绘了一致性hash的基本原理：



一致性hash从根本上来讲是一个键值映射结构 – 它把键（通常是hash过的）映射到物理节点。键经过hash之后的取值空间是一个有序的定长二进制字符串，很显然每个在此范围内的键都会被映射到图A中A、B、C三个节点中的某一个。为了副本复制，将取值空间闭合成一个环，沿环顺时针前行直到所有副本都被映射到合适的节点上，如图B所示。换句话说，Y将被定位在节点B上，因为它在B的范围内，第一个副本应该放置在C，第二个副本放置在A，以此类推。

这种结构的好处体现在增加或减少一个节点的时候，因为它只会引起临接区域的数据重新均衡。如图C所示，节点D的加入只会对数据项X产生影响而对Y无影响。同样，移除节点B（或者B失效）只会影响Y和X的副本，而不会对X自身造成影响。但是，这种做法在带来好处的同时也有弱点，那就是重新均衡的负担都由邻节点承受了，它们将移动大量的数据。通过将每个节点映射到多个范围而不是一个范围可以一定程度上减轻这个问题带来的不利影响，如图D所示。这是一个折中，它避免了重新均衡数据时负载过于集中，但是与基于模块的映射相比，保持了总均衡数量适当降低。

给大规模的集群维护一个完整连贯的hash环很不容易。对于相对小一点的数据库集群就不会有问题，研究如何在对等网络中将数据放置与网络路由结合起来很有意思。一个比较好的例子是Chord算法，它使环的完整性让步于单个节点的查找效率。Chord算法也使用了环映射键到节点的理念，在这方面和一致性hash很相似。不同的是，一个特定节点维护一个短列表，列表中的节点在环上的逻辑位置是指数增长的（如下图）。这使得可以使用二分搜索只需要几次网络跳跃就可以定位一个键。



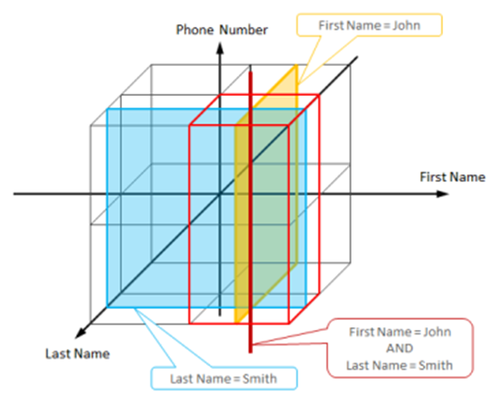
这张图画的是一个由16个节点组成的集群，描绘了节点A是如何查找放在节点D上的key的。 (A) 描绘了路由，(B) 描绘了环针对节点A、B、C的局部图像。在参考资料中有更多关于分散式系统中的数据复制的内容。

**按照多个属性的数据分片**

当只需要通过主键来访问数据的时候，一致性hash的数据放置策略很有效，但是当需要按照多个属性来查询的时候事情就会复杂得多。一种简单的做法（MongoDB使用的）是用主键来分布数据而不考虑其他属性。这样做的结果是依据主键的查询可以被路由到接个合适的节点上，但是对其他查询的处理就要遍历集群的所有节点。查询效率的不均衡造成下面的问题：

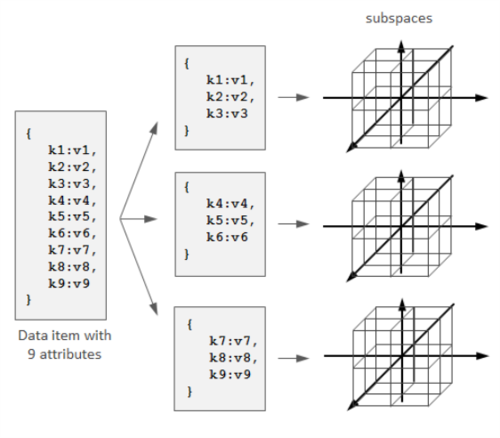
有一个数据集，其中的每条数据都有若干属性和相应的值。是否有一种数据分布策略能够使得限定了任意多个属性的查询会被交予尽量少的几个节点执行？

HyperDex数据库提供了一种解决方案。基本思想是把每个属性视作多维空间中的一个轴，将空间中的区域映射到物理节点上。一次查询会被对应到一个由空间中多个相邻区域组成的超平面，所以只有这些区域与该查询有关。让我们看看参考资料中的一个例子：



每一条数据都是一条用户信息，有三个属性First Name 、Last Name 和Phone Number。这些属性被视作一个三维空间，可行的数据分布策略是将每个象限映射到一个物理节点。像“First Name = John”这样的查询对应到一个贯穿4个象限的平面，也即只有4个节点会参与处理此次查询。有两个属性限制的查询对应于一条贯穿两个象限的直线，如上图所示，因此只有2个节点会参与处理。

这个方法的问题是空间象限会呈属性数的指数函数增长。结果就会是，只有几个属性限制的查询会投射到许多个空间区域，也即许多台服务器。将一个属性较多的数据项拆分成几个属性相对较少的子项，并将每个子项都映射到一个独立的子空间，而不是将整条数据映射到一个多维空间，这样可以一定程度上缓解这个问题：

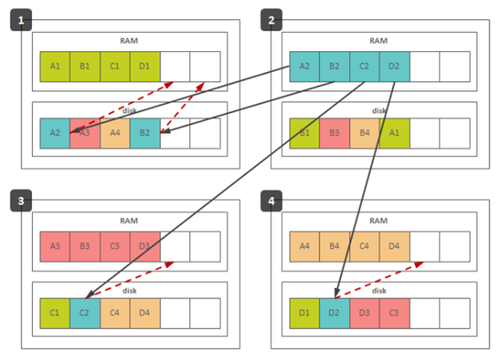


这样能够提供更好的查询到节点的映射，但是增加了集群协调的复杂度，因为这种情况下一条数据会散布在多个独立的子空间，而每个子空间都对应各自的若干个物理节点，数据更新时就必须考虑事务问题。

**钝化副本**

有的应用有很强的随机读取要求，这就需要把所有数据放在内存里。在这种情况下，将数据分片并把每个分片主从复制通常需要两倍以上的内存，因为每个数据都要在主节点和从节点上各有一份。为了在主节点失效的时候起到代替作用，从节点上的内存大小应该和主节点一样。如果系统能够容忍节点失效的时候出现短暂中断或性能下降，也可以不要分片。

下面的图描绘了4个节点上的16个分片，每个分片都有一份在内存里，副本存在硬盘上：



灰色箭头突出了节点2上的分片复制。其他节点上的分片也是同样复制的。红色箭头描绘了在节点2失效的情况下副本怎样加载进内存。集群内副本的均匀分布使得只需要预留很少的内存就可以存放节点失效情况下激活的副本。在上面的图里，集群只预留了1/3的内存就可以承受单个节点的失效。特别要指出的是副本的激活（从硬盘加载入内存）会花费一些时间，这会造成短时间的性能下降或者正在恢复中的那部分数据服务中断。

**系统协调**

在这部分我们将讨论与系统协调相关的两种技术。分布式协调是一个比较大的领域，数十年以来有很多人对此进行了深入的研究。这篇文章里只涉及两种已经投入实用的技术。关于分布式锁，consensus协议以及其他一些基础技术的内容可以在很多书或者网络资源中找到。

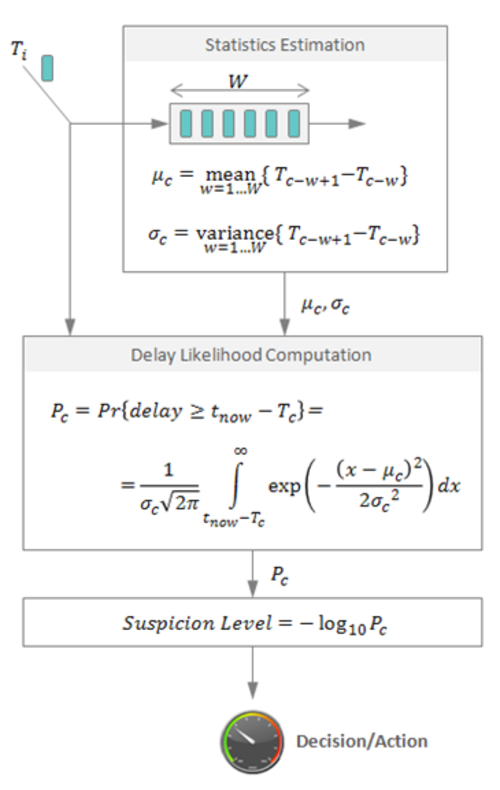
**故障检测**

故障检测是任何一个拥有容错性的分布式系统的基本功能。实际上所有的故障检测协议都基于心跳通讯机制，原理很简单，被监控的组件定期发送心跳信息给监控进程（或者由监控进程轮询被监控组件），如果有一段时间没有收到心跳信息就被认为失效了。除此之外，真正的分布式系统还要有另外一些功能要求：

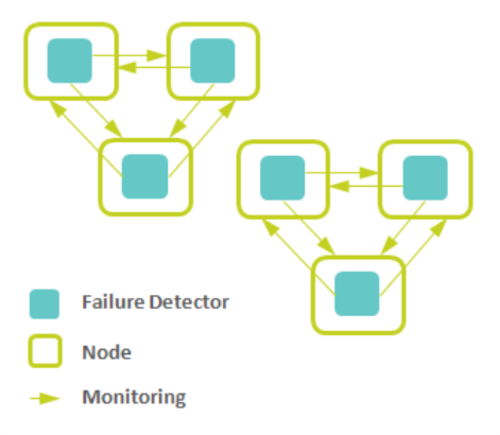
* **自适应。**故障检测应该能够应对暂时的网络故障和延迟，以及集群拓扑、负载和带宽的变化。但这有很大难度，因为没有办法去分辨一个长时间没有响应的进程到底是不是真的失效了，因此，故障检测需要权衡故障识别时间（花多长时间才能识别一个真正的故障，也即一个进程失去响应多久之后会被认为是失效）和虚假警报率之间的轻重。这个权衡因子应该能够动态自动调整。
* **灵活性。**乍看上去，故障检测只需要输出一个表明被监控进程是否处于工作状态的布尔值，但在实际应用中这是不够的。我们来看参考资料中的一个类似MapReduce的例子。有一个由一个主节点和若干工作节点组成的分布式应用，主节点维护一个作业列表，并将列表中的作业分配给工作节点。主节点能够区分不同程度的失败。如果主节点怀疑某个工作节点挂了，他就不会再给这个节点分配作业。其次，随着时间推移，如果没有收到该节点的心跳信息，主节点就会把运行在这个节点上的作业重新分配给别的节点。最后，主节点确认这个节点已经失效，并释放所有相关资源。
* **可扩展性和健壮性。**失败检测作为一个系统功能应该能够随着系统的扩大而扩展。他应该是健壮和一致的，也即，即使在发生通讯故障的情况下，系统中的所有节点都应该有一个一致的看法（即所有节点都应该知道哪些节点是不可用的，那些节点是可用的，各节点对此的认知不能发生冲突，不能出现一部分节点知道某节点A不可用，而另一部分节点不知道的情况）

**所谓的累计失效检测器可以解决前两个问题，Cassandra对它进行了一些修改并应用在产品中。其基本工作流程如下：**

* 对于每一个被监控资源，检测器记录心跳信息到达时间Ti。
* 计算在统计预测范围内的到达时间的均值和方差。
* 假定到达时间的分布已知（下图包括一个正态分布的公式），我们可以计算心跳延迟（当前时间t\_now和上一次到达时间Tc之间的差值） 的概率，用这个概率来判断是否发生故障。可以使用对数函数来调整它以提高可用性。在这种情况下，输出1意味着判断错误（认为节点失效）的概率是10%，2意味着1%，以此类推。



根据重要程度不同来分层次组织监控区，各区域之间通过谣言传播协议或者中央容错库同步，这样可以满足扩展性的要求，又可以防止心跳信息在网络中泛滥。如下图所示（6个故障检测器组成了两个区域，互相之间通过谣言传播协议或者像ZooKeeper这样的健壮性库来联系）：



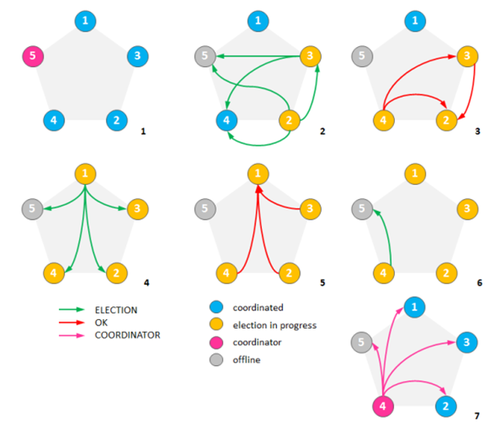
**协调者竞选**

协调者竞选是用于强一致性数据库的一个重要技术。首先，它可以组织主从结构的系统中主节点的故障恢复。其次，在网络隔离的情况下，它可以断开处于少数的那部分节点，以避免写冲突。

Bully 算法是一种相对简单的协调者竞选算法。MongoDB 用了这个算法来决定副本集中主要的那一个。Bully 算法的主要思想是集群的每个成员都可以声明它是协调者并通知其他节点。别的节点可以选择接受这个声称或是拒绝并进入协调者竞争。被其他所有节点接受的节点才能成为协调者。节点按照一些属性来判断谁应该胜出。这个属性可以是一个静态ID，也可以是更新的度量像最近一次事务ID（最新的节点会胜出）。

下图的例子展示了bully算法的执行过程。使用静态ID作为度量，ID值更大的节点会胜出：

1. 最初集群有5个节点，节点5是一个公认的协调者。
2. 假设节点5挂了，并且节点2和节点3同时发现了这一情况。两个节点开始竞选并发送竞选消息给ID更大的节点。
3. 节点4淘汰了节点2和3，节点3淘汰了节点2。
4. 这时候节点1察觉了节点5失效并向所有ID更大的节点发送了竞选信息。
5. 节点2、3和4都淘汰了节点1。
6. 节点4发送竞选信息给节点5。
7. 节点5没有响应，所以节点4宣布自己当选并向其他节点通告了这一消息。



协调者竞选过程会统计参与的节点数目并确保集群中至少一半的节点参与了竞选。这确保了在网络隔离的情况下只有一部分节点能选出协调者（假设网络中网络会被分割成多块区域，之间互不联通，协调者竞选的结果必然会在节点数相对比较多的那个区域中选出协调者，当然前提是那个区域中的可用节点多于集群原有节点数的半数。如果集群被隔离成几个区块，而没有一个区块的节点数多于原有节点总数的一半，那就无法选举出协调者，当然这样的情况下也别指望集群能够继续提供服务了）。