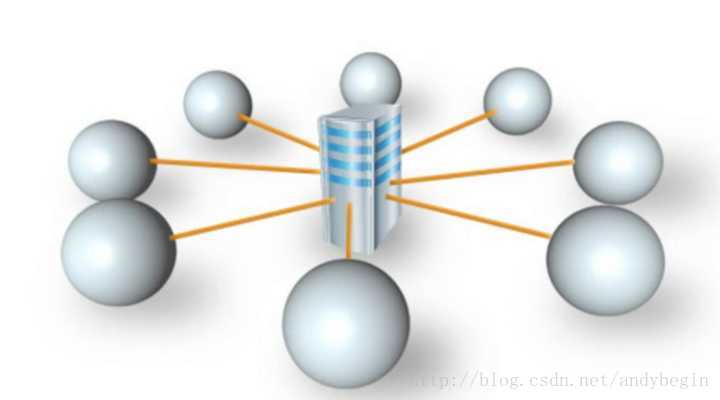
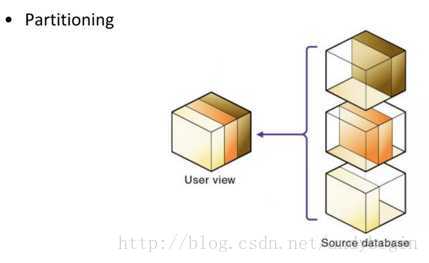
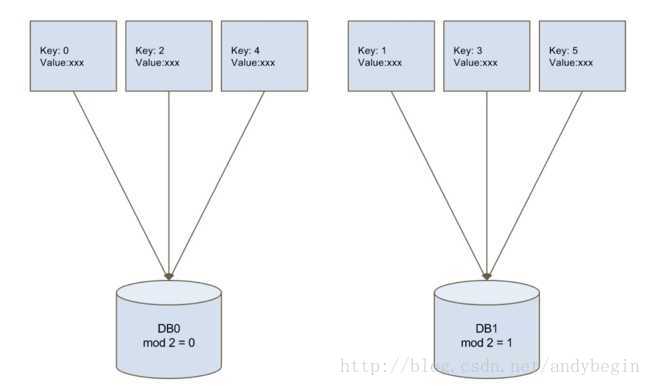
# **cassandra 架构**

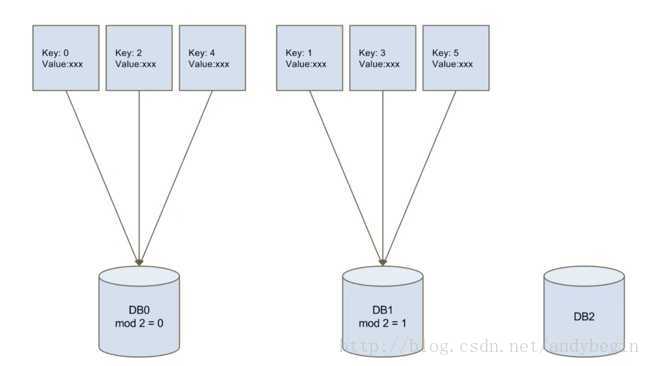
https://blog.csdn.net/andybegin/article/details/78490027

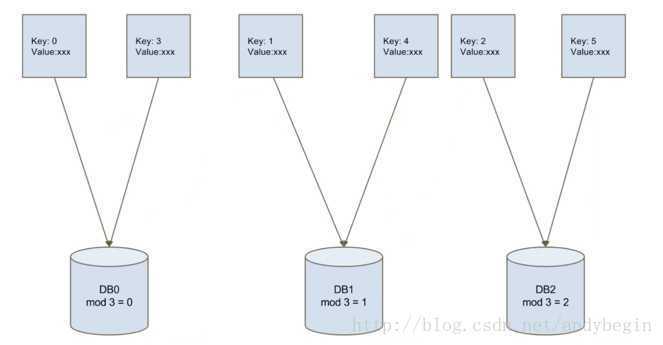
所有系统最开始都是centralized。但是有很多缺点，比如单机的处理能力不够。

Centralized database如下图：   


所以出现了分布式的数据库。如下图：   


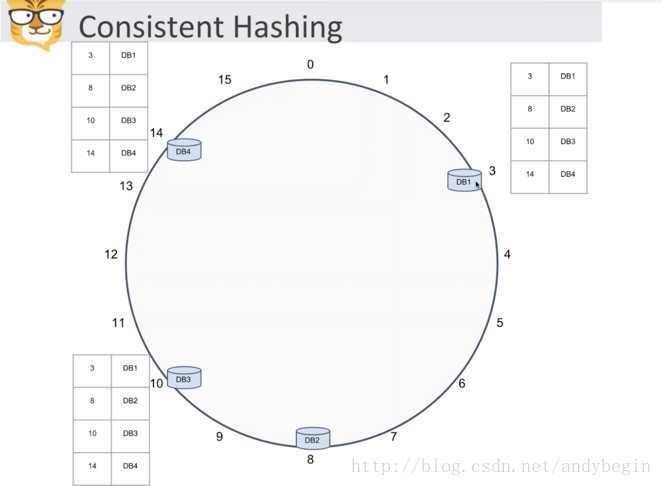
Partitioning的方式有range partitioning, list partitioning, hash partitioning。其中hash partitioning使用最多。如下图，数据根据key的hash结果，存入不同的数据库中。相比其他两种，它能够balance，保证每个数据库存的数据量相当。   


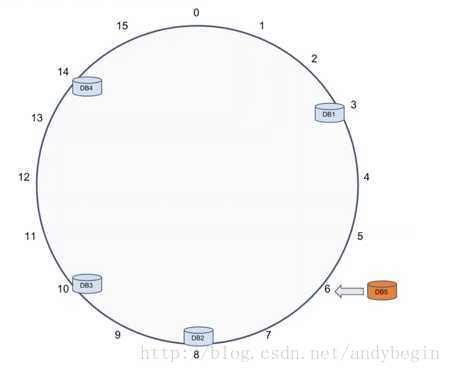
如果要加一个server，如下图，要将key重新hash，实现新的balance，这需要O(n)级别的移动。这是这种方法的缺点。   




如果hash function和server数不挂钩，就能解决问题。一个方法如下图，有2的n次方个key space，每个server负责一部分key space（key的value）。

有四台server，每个server占在一个数上，负责处理这个数之前的数。比如DB1负责15，0，1，2，3。这个结构是peer-to-peer，没有主从关系。

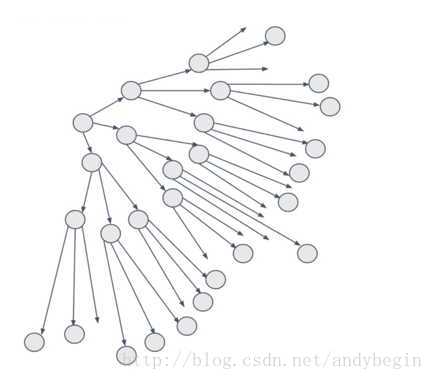
每个server都可以作为入口处理client的请求。Client找到任一个server，数据的key经过hash，得到12。DB1找它的表，找到第一个大于这个值（12）的数，此例中是14，对应DB4，于是数据被存入DB4。Client索要数据的过程和存储一致。这种方式的好处在于添加server和减少server时，数据迁移大大减小。   


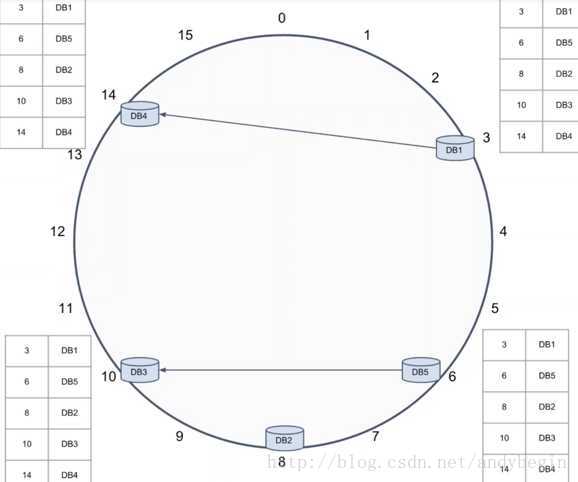
假如希望把新server DB5放在环中6处，这意味着以后4，5，6都归这个新server管理。4，5，6原来的数据存在DB2，这些数据需要存到DB5。只需要和一个点通信。M个server，N个key space，只需N/M级别时间。假设DB5需要移除，只需要把4，5，6还给DB2。任何节点加入，只需要向后节点要数据，任何节点离开，只需要将数据给后节点。   


# **节点通信**

在peer-to-peer结构中有一个很有名的协议，叫Gossip Protocol。它的精髓在于一传十，十传百。

每个node有个neighbor list，收到信息后转发给neighbor list里的node。node之前收到过消息则不再转发，降低网络overhead。

它两个重要参数，一个是Fan number，表示每个node转发的node数量，此例中是3，第二个是Time to live（TTL）。下图中，起始node的参数是3，它传给的node的参数变为2，依次递减，当node的参数是0时，不再转发。   


比如之前例子中，DB5加入后，将加入信息传给其他server，DB5先将消息告诉DB1，DB3然后DB1将消息告诉DB4。如下图：   


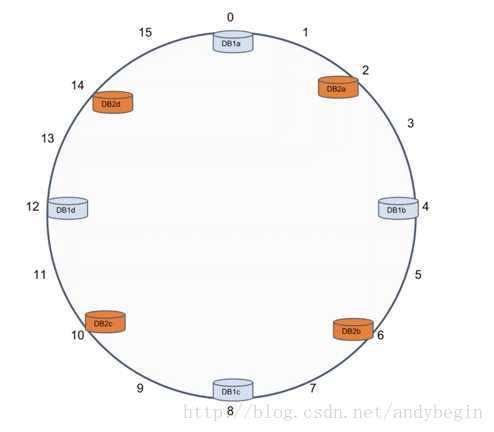
下图是Data Model，数据要有primary key：   


将primary key进行hash之后，将属于key的这一行数据作为值存到server。Cassandra只基于行存储。   

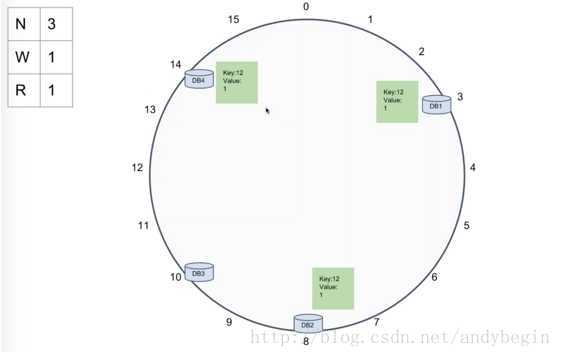

现在有load balance的问题，因为即使最开始数据分布比较均匀，在加入新的server之后，还是会使有的server处理的数据很少，有的server处理的数据很多。此外，数据量小时，很容易出现hash后某些key多，某些key少。另一个不balance的地方在于单机处理数据的不同。

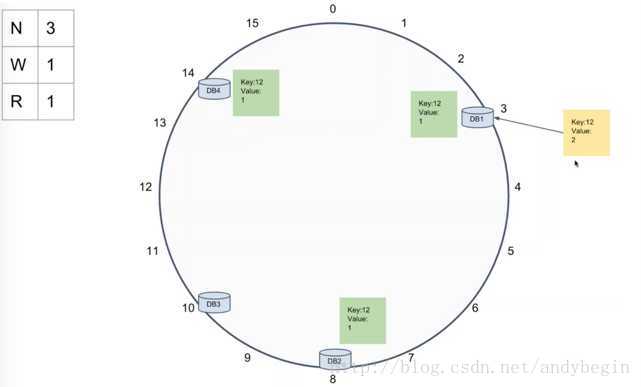
解决方案有两个：virtual nodes和move nodes。

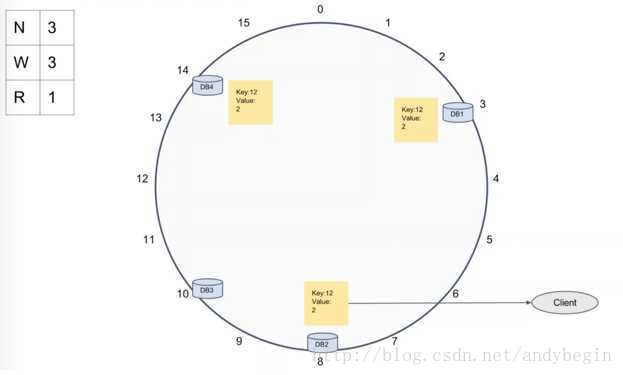
## **Virtual node**

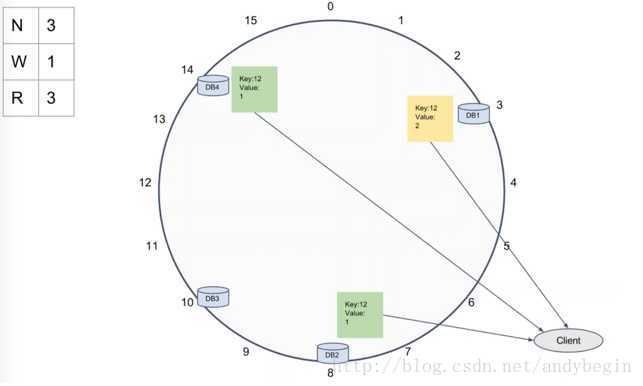
Virtual node指一台机器能充当几个点。比如有两台机器，DB1和DB2，它们分到相同数据量的几率很小，但是如果每个server可以扮演多个点，如下图，DB1扮演DB1a，DB1b，DB1c，DB1d。这两个server分到相同数据量的几率就大很多。   


目前为止，每个节点负责一部分数据，所以需要数据replication。备份需要注意的是不要将所有的node存在相同的physical server上。数据备份带来consistency的问题。cassandra是一个AP系统，遵循CAP（C：数据是一致的，A：数据是可用的，P：数据被切分）原则。

****它有tunable consistency，有三个参数可调：N代表备份数，默认为3，W是写操作所需的成功数，R是读操作所需的成功数。****下图是一个例子：   


N为3，最简单的方法是用此DB后两个DB作为备份。DB4的数据在DB1和DB4备份。此时再写一个数。W为1，代表只要给一个DB写成功就任务整个新数据的写成功。R为1，代表只要从一个DB读到需要数据就算读取成功，但这样的问题在于读取的数据不一定是最新的。   


下面的例子是W为3。三个DB都写成功才算是数据的写成功。   


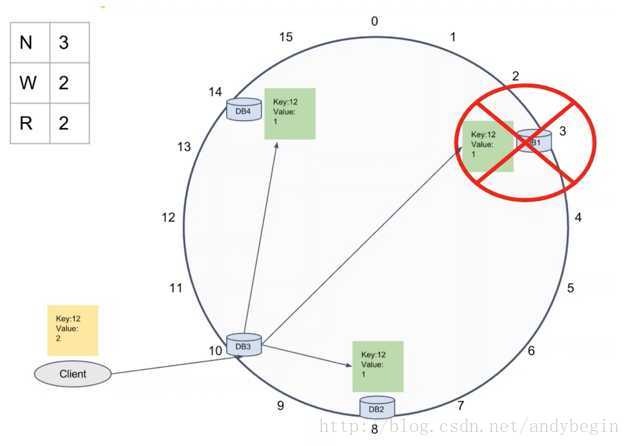
如果W为1，R为3，也能保证读到最新的数据。   


****Strong Consistency保证一定读到新数据：W + R > N。****   
如果W少R多时，判断哪个数据是新数据时，可以通过比较timestamp，数据的存储有timestamp信息。

# **最终一致性**

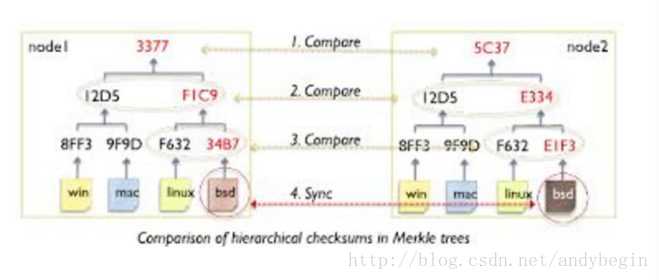
有三种方案：read repair，hinted handoff，anti-entropy repair

1）read repair   
可以设置一个概率来触发read repair，当它被触发时，向所有replica发请求，从一个replica读数据，从其他replica读摘要，如果摘要和数据不一致，强迫所有replica进行比较，同步数据。

2）hinted handoff   
即使W为2，也会尝试给所有机器写。当DB1 fail，如果有两个其他机器写成功，仍会反应写成功。DB3本不需要存这个数据，但现在存储这个数据，并且在下面写一个hint：DB1，表示数据本该存在DB1里。在短时间内，DB3会不断尝试把数据写回DB1。   


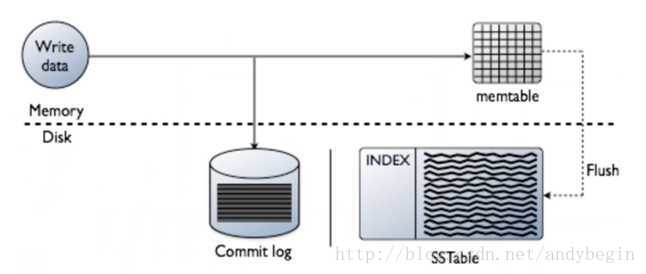
3）anti-entropy repair   
每个node定期和其他replica进行数据比较，如果发现数据不同，进行merge。定期的时间默认为一秒钟。如果node fail，重新启动会尝试和replica同步。如果数据很久没有被读取，当它被读取时，进行同步数据更新。

# **检查数据不一致性**

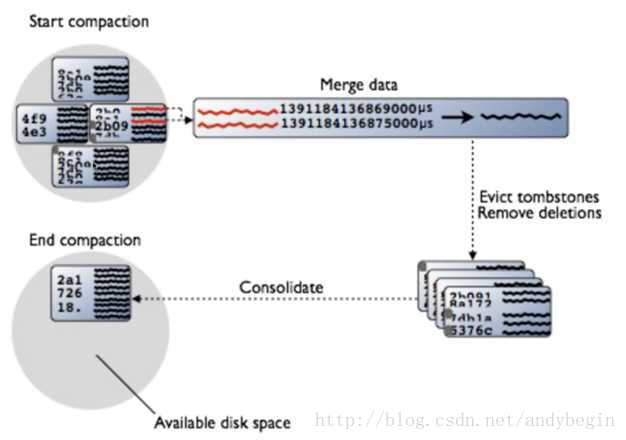
Cassandra使用merkle trees的方式探测inconsistency。如下图：   


它将每块数据进行hash。把每两个hash后结果进行hash，依次类推，形成一个二叉树。当发现不同时，类似二叉树查找进行比较。它的缺点是如果写操作很多时，merkle tree的构建非常复杂。

# **提高读写效率**

1) increase write speed   


写的时候，首先尝试往内存里写，同时写一个log。当内存里的table写满后，存入disk。

2) increase read speed   


使用compaction。一定情况下，将几个数据块进行merge。把同一个key的数据块放在一起。同时可以设置TTL，这个TTL以时间为单位，时间一到，数据删除，减少merge。

Compaction提供三个方案：   
1）SizeTieredCompactionStrategy (STCS)：每四个数据块压一块，对于insert多的系统好。   
2）LeveledCompactionStrategy (LCS)：对于update多的系统好   
3）DataTieredCompactionStrategy (DTCS)：根据时间进行compaction