

从图中可以看出，MongoDB sharding 主要分为 3 大部分。shard 节点、config 节点和 config 节点。

对客户端来说，直接访问的是 图中绿色的 mongos 节点。背后的 config 节点和 shard 节点是客户端不能直接访问的。

mongos 的主要作用是数据路由。从元数据中定位数据位置，合并查询结果。另外，mongos 节点还负责数据迁移和数据自动平衡，并作为 sharding 集群的管理节点。它对外的接口就和普通的 mongod 一样。因此，可以使用标准 mongodb 客户端和驱动进行访问。mongos 节点是无状态的，本身不保存任何数据和元数据，因此可以任意水平扩展，这样任意一个节点发生故障都可以很容易的进行故障转移，不会造成严重影响。

其中蓝色的 shard 节点就是实际存放数据的数据节点。每个 shard 节点可以是单个 mongod 实例，也可以使一个 replica set 。通常在使用 sharding 的时候，都会同时使用 replica set 来实现高可用，以免集群内有单个节点出故障的时候影响服务，造成数据丢失。同时，可以进一步通过读写分离来分担负载。对于每个开启 sharding 的 db 来说，都会有一个 默认 shard 。初始时，第一个 chunk 就会在那里建立。新数据也就会先插入到那个 shard 节点中去。

图中紫色的 config 节点存储了元数据，包括数据的位置，即哪些数据位于哪些节点，以及集群配置信息。config 节点也是普通的 mongod 。如图所示，一组 config 节点由 3 个组成。这 3 个 config 节点并非是一个 replica set。它们的数据同步是由 mongos 执行两阶段提交来保证的。这样是为了避免复制延迟造成的元数据不同步。config 节点一定程度上实现了高可用。在一个或两个节点发生故障时，config 集群会变成只读。但此时，整个 sharding 集群仍然可以正常读写数据。只是无法进行数据迁移和自动均衡而已。

config 节点里存放的元数据都有些啥呢？连上 mongos 后，

use config; show collections

结果是

settings

shards

databases

collections

chunks

mongos

changelog

还可以进一步查看这些东西的数据。这个 config 库就是后端 config 节点上的数据的映射，提供了一个方便的读取元数据的入口。这些 collection 里面都是什么呢？ settings 里是 sharding 的配置信息，比如数据块大小，是否开启自动平衡。shards 里存放的是后端 shard 节点的信息，包括 ip，端口等。databases 里存放的是数据库的信息，是否开启 sharding，默认 shard 等。collections 中则是哪些 collection 启用了 sharding，已经用了什么 shard key。chunks 里是数据的位置，已经每个 chunk 的范围等。mongos 里是关于 mongos 的信息，changelog 是一个 capped collection，保存了最近的 10m 元数据变化记录。

mongodb sharding 的搭建也很容易。简单的几步就能完成。

先启动若干 shard 节点

mongod --shardsvr

启动 3 个 config 节点

mongod --configsvr

启动 mongos

mongos --configdb=192.168.1.100, 192.168.1.101, 192.168.1.102

这里，–shardsvr 参数只起到修改默认端口为 27018 作用，–configsvr 则修改默认端口为 27019 以及默认路径为 /data/configdb。此外并没有什么直接作用。实际使用时，也可以自己指定端口和数据路径。此外，这两个参数的另一个作用就是对进程进行标记， 这样在 ps aux 的进程列表里，就很容易确定进程的身份。–configdb 参数就是 config 节点的地址。如果更改了默认端口，则需要在这里加上。

然后我们把数据节点加入集群：在 mongos 上运行

use admin

sh.addShard(’ [hostname]:[port]’)

如果使用的事 replicaSet，则是

use admin

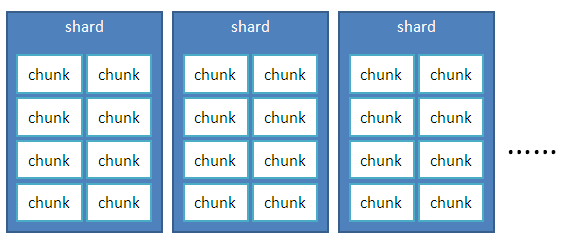
sh.addShard(’replicaSetName/,,’)

接着就是启用 sharding 了。

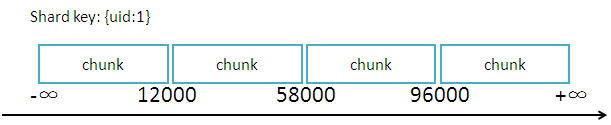
sh.enableSharding(dbname)

sh.shardCollection(fullName, key, unique)

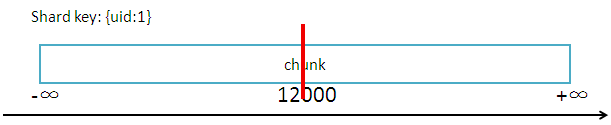
这样就可以了。还是很简单的吧。如果 collection 里有数据，则会自动进行数据平衡。



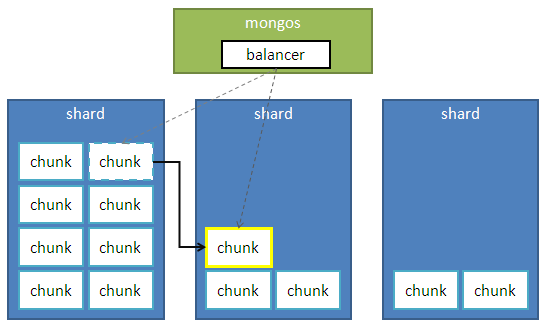
之前说过，mongodb 的 sharding 把数据分成了数据块（chunk）来进行管理。现在来看看 chunk 究竟是怎么回事。在 mongodb sharding 中，chunk 是数据迁移的基本单位。每个节点中的数据都被划分成若干个 chunk 。一个 chunk 本质上是 shard key 的一个连续区间。chunk 实际上是一个逻辑划分而非物理划分。sharding 的后端就是普通的 mongod 或者 replica set，并不会因为是 sharding 就对数据做特殊处理。一个 chunk 并不是实际存储的一个页或者一个文件之类，而是仅仅在 config 节点中的元数据中体现。mongodb 的sharding 策略实际上就是一个 range 模式。

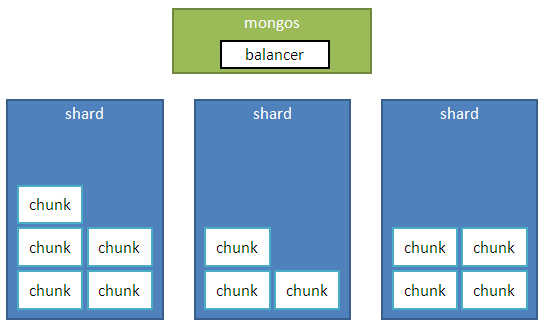


如图，第一个 chunk 的范围就是 uid 从 -∞ 到 12000 范围内的数据。第二个就是 12000 到 58000 。以此类推。对于一个刚配置为 sharding 的 collection ，最开始只有一个 chunk，范围是从 -∞ 到 +∞。



随着数据的增长，其中的数据大小超过了配置的 chunk size，默认是 64M 则这个 chunk 就会分裂成两个。因为 chunk 是逻辑单元，所以分裂操作只涉及到元数据的操作。数据的增长会让 chunk 分裂得越来越多。这时候，各个 shard 上的 chunk 数量就会不平衡。这时候，mongos 中的一个组件 balancer 就会执行自动平衡。把 chunk 从 chunk 数量最多的 shard 节点挪动到数量最少的节点。



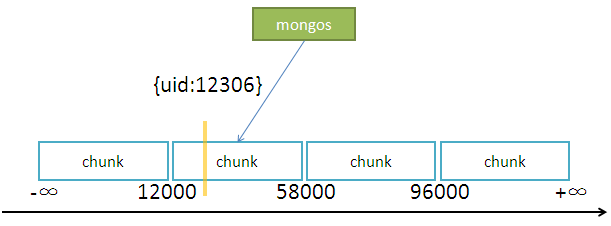


最后，各个 shard 节点上的 chunk 数量就会趋于平衡。当然，balance 不一定会使数据完全平均，因为移动数据本身有一定成本，同时为了避免极端情况下早晨数据来回迁移，只有在两个 shard 的 chunk 数量之差达到一定阈值时才会进行。默认阈值是 8 个。也就是说，默认情况下，只有当两个节点的数据量差异达到 64M \* 8 == 256M 的时候才会进行。这样就不用对刚建好的 sharding ，插入了不少数据，为什么还是都在一个节点里感到奇怪了。那只是因为数据还不够多到需要迁移而已。

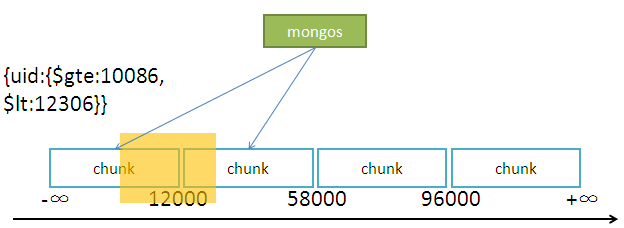
在数据迁移的过程中，仍然可以进行数据读写，并不会因此而影响可用性。那么 mongodb 是怎么做到的呢？在数据迁移过程中，数据读写操作首先在源数据节点中进行。待迁移完毕后，再将这期间的更新操作同步到新节点中去。最后再更新 config 节点，标记数据已经在新的地方，完成迁移。只有在最后同步迁移期间的操作的时候，需要锁定数据更新。这样就讲锁定时间尽可能缩小，大大降低数据迁移对服务 的影响。

mongodb 的 sharding 和传统 sharding 的最大区别就在于引入了元数据。看似增加了复杂度，并增加了一些额外的存储，但是由此带来的灵活性却是显而易见的。传统的 sharding 本质上是对数据的静态映射，所有那些数据迁移的困难都是由此而来。而引入元数据以后，就变静态映射为动态映射。数据迁移就不再是难事了。从而从根本上解决 了问题。另一方面，用元数据实现 chunk 则降低了实现难度，后端节点仍然可以使用原有的技术。同时，因为不需要对后端数据进行变动，也使部署迁移变得更容易，只需要另外加上 mongos 节点和 config 节点即可。

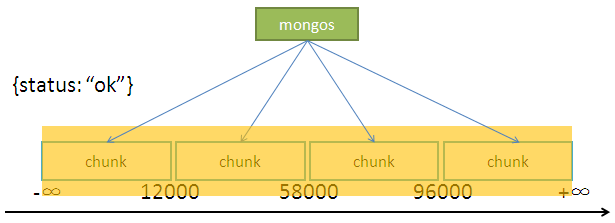
再说说数据路由功能。mongos 的最主要功能就是作为数据路由，找到数据的位置，合并查询结果。来看看它是如何处理的。如果查询的条件是 shard key ，那么 mongos 就能从元数据直接定位到 chunk 的位置，从目标节点找到数据。



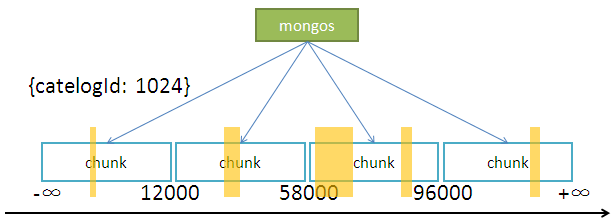
如果查询条件是 shard key 的范围，由于 chunk 是按 shard key 的范围来划分的，所以 mongos 也可以找到数据对应 chunk 的位置，并把各个节点返回的数据合并。



如果查询的条件不是任何一个索引，原来的全 collection 遍历仍然不可避免。但是会分发到所有节点进行。所以，还是可以起到分担负载的作用。



如果查询的条件是一个索引，但不是 shard key，查询也会被分发到所有节点，不过在每个节点上索引仍然有效。



如果是按查询 shard key 进行排序，同样由于 chunk 是一个 shard key 的范围，则会依次查询各 chunk 所在节点，而无需返回所有数据再排序。如果不是按 shard key 排序，则会在每个节点上执行排序操作，然后由 mongos 进行归并排序。由于是对已排序结果的归并排序，所以在 mongos 上不会有多少压力，查询结果的游标也会变成在每个节点上的游标。并不需要把所有数据都吐出来。

从上面可以看到，对 sharding 集群来说，shard key 的选择是至关重要的。shard key 其实就相当于数据库的聚簇索引，所以选择聚簇索引的原则和选择 shard key 的原则是差不多的。同样， shard key 一旦设定就无法再更改，所以，选择的时候就要谨慎。shard key 的选择主要就这么几点。

首先，shard key 的值要是固定的，不会被更改的。因为一旦这个值被更改，就有可能会从一个节点被挪动到另一个节点，从而带来很大的开销。

第二，shard key 要有足够的区分度。同样因为 chunk 是一个 shard key 的范围，所以 shard key 相同的值只能位于同一个 chunk 。如果 shard key 相同的值很大，致使一个 chunk 的大小超过了 chunk size，也无法对 chunk 进行分裂，数据均衡。同时，和一般的数据库索引一样，更好的区分度也能提高查询性能。

第三，shard key 还要有一定的随机性而不是单向增长。单向增长的 shard key 会导致新插入的数据都位于一个 chunk 中，在在某一个 shard 节点中产生集中的写压力。所以，最好避免直接使用 \_id ，时间戳 这种单向增长的值作为 shard key。

mongodb 的 sharding 有很多优势，但是也同样有其局限性。

首先，mongodb 只提供了 range 模式的 sharding。这种模式虽然可以对按 shard key进行 range 查询、排序进行优化，但是也会造成使用单向增长的值时，写入集中的结果。

第二，启用了 sharding 之后，就无法保证除 shard key 以为其他的索引的唯一性。即使设为 unique，也只是保证在每个节点中唯一。有一个办法是，把索引设为 {<shard\_key>:1, <unique\_key>:1} 。但是这样并不一定满足业务逻辑需求。

第三，启用 sharding 后，无法直接使用 group() 。但是可以用 map reduce 功能作为替代。

第四，虽然数据迁移操作对读写影响很小，但是这个过程需要先把数据从磁盘中换入内存才能进行，所以可能会破坏热数据缓存。此外，数据迁移也还是会增大 io 压力，所以可以考虑平时关闭自动平衡，在凌晨压力小的时候再进行。

最后，config 节点的元数据同步对时钟准确性要求比较高，一旦各 config 时钟误差大了，就会出现无法上锁，从而无法更改，导致数据集中。因此 ntp 时钟同步时必不可少的。

在这里再说一下 sharding 集群的备份问题。由于后端数据节点仍然是普通的 mongod 或 replica set，所以备份其实和原先差不多。只是需要注意的是，备份前需要停止自动平衡，保证备份期间 sharding 的元数据不会变动，然后备份 shard 节点和 config 节点数据即可。