## 一、垂直拆分与水平拆分

 Sharding的基本思想就要把一个**[数据库](http://lib.csdn.net/base/14" \o "MySQL知识库" \t "http://blog.csdn.net/bluishglc/article/details/_blank)**切 分成多个部分放到不同的数据库(server)上，从而缓解单一数据库的性能问题。不太严格的讲，对于海量数据的数据库，如果是因为表多而数据多，这时候 适合使用垂直切分，即把关系紧密（比如同一模块）的表切分出来放在一个server上。如果表并不多，但每张表的数据非常多，这时候适合水平切分，即把表 的数据按某种规则（比如按ID散列）切分到多个数据库(server)上。当然，现实中更多是这两种情况混杂在一起，这时候需要根据实际情况做出选择，也 可能会综合使用垂直与水平切分，从而将原有数据库切分成类似矩阵一样可以无限扩充的数据库(server)阵列。下面分别详细地介绍一下垂直切分和水平切 分.  
  垂直切分的最大特点就是规则简单，实施也更为方便，尤其适合各业务之间的耦合度非常低，相互影响很小，业务逻辑非常清晰的系统。在这种系统中，可以很容易做到将不同业务模块所使用的表分拆到不同的数据库中。根据不同的表来进行拆分，对应用程序的影响也更小，拆分规则也会比较简单清晰。（这也就是所谓的”share nothing”）。

## 二、切分策略

      如前面所提到的，切分是按先垂直切分再水平切分的步骤进行的。垂直切分的结果正好为水平切分做好了铺垫。垂直切分的思路就是分析表间的聚合关系，把关系紧 密的表放在一起。多数情况下可能是同一个模块，或者是同一“聚集”。这里的“聚集”正是领域驱动设计里所说的聚集。在垂直切分出的表聚集内，找出“根元 素”（这里的“根元素”就是领域驱动设计里的“聚合根”），按“根元素”进行水平切分，也就是从“根元素”开始，把所有和它直接与间接关联的数据放入一个 shard里。这样出现跨shard关联的可能性就非常的小。应用程序就不必打断既有的表间关联。比如：对于社交网站，几乎所有数据最终都会关联到某个用 户上，基于用户进行切分就是最好的选择。再比如论坛系统，用户和论坛两个模块应该在垂直切分时被分在了两个shard里，对于论坛模块来说，Forum显 然是聚合根，因此按Forum进行水平切分，把Forum里所有的帖子和回帖都随Forum放在一个shard里是很自然的。  
  
      对于共享数据数据，如果是只读的字典表，每个shard里维护一份应该是一个不错的选择，这样不必打断关联关系。如果是一般数据间的跨节点的关联，就必须打断。

**需要特别说明的是： 当同时进行垂直和水平切分时，切分策略会发生一些微妙的变化。比如：在只考虑垂直切分的时候，被划分到一起的表之间可以保持任意的关联关系，因此你可以按 “功能模块”划分表格，但是一旦引入水平切分之后，表间关联关系就会受到很大的制约，通常只能允许一个主表（以该表ID进行散列的表）和其多个次表之间保 留关联关系，也就是说：当同时进行垂直和水平切分时， 在垂直方向上的切分将不再以“功能模块”进行划分，而是需要更加细粒度的垂直切分，而这个粒度与领域驱动设计中的“聚合”概念不谋而合，甚至可以说是完全 一致，每个shard的主表正是一个聚合中的聚合根！这样切分下来你会发现数据库分被切分地过于分散了（shard的数量会比较多，但是shard里的表 却不多），为了避免管理过多的数据源，充分利用每一个数据库服务器的资源，可以考虑将业务上相近，并且具有相近数据增长速率（主表数据量在同一数量级上）的两个或多个shard放到同一个数据源里，每个shard依然是独立的，它们有各自的主表，并使用各自主表ID进行散列，不同的只是它们的散列取模（即节点数量）必需是一致的。（**

本文着重介绍sharding的基本思想和理论上的切分策略，关于更加细致的实施策略和参考事例请参考我的另一篇博文：[数据库分库分表(sharding)系列(一) 拆分实施策略和示例演示](http://blog.csdn.net/bluishglc/article/details/7696085" \t "http://blog.csdn.net/bluishglc/article/details/_blank)

）

1.事务问题：  
解决事务问题目前有两种可行的方案：分布式事务和通过应用程序与数据库共同控制实现事务下面对两套方案进行一个简单的对比。  
方案一：使用分布式事务  
    优点：交由数据库管理，简单有效  
    缺点：性能代价高，特别是shard越来越多时  
方案二：由应用程序和数据库共同控制  
     原理：将一个跨多个数据库的分布式事务分拆成多个仅处  
           于单个数据库上面的小事务，并通过应用程序来总控  
           各个小事务。  
     优点：性能上有优势  
     缺点：需要应用程序在事务控制上做灵活设计。如果使用     
           了spring的事务管理，改动起来会面临一定的困难。  
2.跨节点Join的问题  
      只要是进行切分，跨节点Join的问题是不可避免的。但是良好的设计和切分却可以减少此类情况的发生。解决这一问题的普遍做法是分两次查询实现。在第一次查询的结果集中找出关联数据的id,根据这些id发起第二次请求得到关联数据。  
  
3.跨节点的count,order by,group by以及聚合函数问题  
      这些是一类问题，因为它们都需要基于全部数据集合进行计算。多数的代理都不会自动处理合并工作。解决方案：与解决跨节点join问题的类似，分别在各个节 点上得到结果后在应用程序端进行合并。和join不同的是每个结点的查询可以并行执行，因此很多时候它的速度要比单一大表快很多。但如果结果集很大，对应 用程序内存的消耗是一个问题。

## 三、垂直拆分的粒度

垂直切分的粒度指的是在做垂直切分时允许几级的关联表放在一个shard里．这个问题对应用程序和sharding实现有着很大的影响．

关 联打断地越多，则受影响的join操作越多，应用程序为此做出的妥协就越大，但单表的路由会越简单，与业务的关联性会越小，就越容易使用统一机制处理．在 此方向上的极端方案是：打断所有连接，每张表都配有路由规则，可以使用统一机制或框架自动处理．比如amoeba这样的框架，它的路由能且仅能通过SQL 的特征（比如某个表的id）进行路由．

反 之，若关联打断地越少，则join操作的受到的限制就小，应用程序需要做出的妥协就越小，但是表的路由就会变复杂，与业务的关联性就越大，就越难使用统一 机制处理，需要针对每个数据请求单独实现路由．在此方向上的极端方案是：所有表都在一个shard里，也就是没有垂直切分，这样就没有关联被打断．当然这 是非常极端的，除非整个**[数据库](http://lib.csdn.net/base/14" \o "MySQL知识库" \t "http://blog.csdn.net/bluishglc/article/details/_blank)**很简单，表的数量很少．

实际的粒度掌控需要结合“业务紧密程度”和“表格数据量”两个因素综合考虑，一般来说：

* 若划归到一起的表格关系紧密，且数据量并不大，增速也非常缓慢，则适宜放在一个shard里，不需要再进行水平切分;
* 若划归到一起的表格数据量巨大且增速迅猛，则势必要在垂直切分的基础上再进行水平切分，水平切分就意味着原单一shard会被细分成多个更小的shard，每一个shard存在一个主表（即会以该表ID进行散列的表）和多个相之相关的关联表。

总之，垂直切分的粒度在两个相反的方向上呈现优势与劣势并存并相互博弈的局面．**[架构](http://lib.csdn.net/base/16" \o "大型网站架构知识库" \t "http://blog.csdn.net/bluishglc/article/details/_blank)**师需要做的是结合项目的实际情况在两者之间取得收益最大化的平衡．