# 一、分库/分表键的选择

## 1、什么是分库/分表键

分库/分表键即分库/分表字段，是在水平拆分过程中用于生成拆分规则的数据表字段。

数据表拆分的首要原则，就是要尽可能找到数据表中的数据在业务逻辑上的主体，并确定大部分（或核心的）数据库操作都是围绕这个主体的数据进行，然后可使用该主体对应的字段作为分表键，进行分库分表。

1）业务逻辑上的主体，通常与业务的应用场景相关，下面的一些典型应用场景都有明确的业务逻辑主体，可用于分表键：

- 面向用户的互联网应用，都是围绕用户维度来做各种操作，那么业务逻辑主体就是用户，可使用用户ID字段作为分表键；
- 侧重于订单的电商应用，都是围绕订单维度来进行各种操作，那么业务逻辑主体就是订单，可使用订单号字段作为分表键；

以此类推，其它类型的应用场景，大多也能找到合适的业务逻辑主体作为分表键的选择。

2）如果确实找不到合适的业务逻辑主体作为分表键，那么可以考虑下面的方法来选择分表键：

- 根据数据分布和访问的均衡度来考虑分表键，尽量将数据表中的数据相对均匀地分布在不同的物理分库/分表中，适用于大量分析型查询的应用场景（查询并发度大部分能维持为1）；
- 按照数字（字符串）类型与时间类型字段相结合作为分表键，进行分库和分表，适用于日志检索类的应用场景。

**注意**：无论选择什么拆分键，采用何种拆分策略，都要注意拆分值是否存在热点的问题，尽量规避热点数据来选择拆分键。

**注意**：不一定需要拿数据库主键当做分表键，也可以拿其他业务值当分表键。拿主键当分表键的好处是可以散列均衡，减少热点问题。

## 2、多个分表键如何处理

大部分场景下，一张表的查询条件比较单一，只需要一个分表键即可；但是有的时候，业务必须要有多个分表键，没有办法归一成一个。此时一般有四种处理方式：

名词定义：

- 主分表键=主维度，在主维度上，数据能够增删改查；
- 辅助分表键=辅维度，在辅助维度上，只能进行数据查询

**1）在主维度上全表扫描**

由于SQL中没有主维度，所以在对辅助维度进行查询时，只能在所有的主维度的表进行查询一遍，然后聚合。如果分了4个库，32张表，最终会以4个线程去并发查询32张表，最终把结果合并输出。

**适用场景**：辅助维度的查询请求的量很小，并且是运营查询或离线查询，对性能要求不高

**2）多维度数据进行冗余同步**

主维度的数据，通过binlog的方式，同步到辅助维度一份。那么在查询辅助维度时，会落到辅助维度的数据上进行查询。

**适用场景**：辅助维度的查询请求的量也很可观，不能直接使用第一种全表扫描的方式

**3）二维巧妙归一维**

辅助维度其实有的时候也是主维度，比如在订单表Order中，OrderID和UserID其实是一一对应的，Order表的主维度是UserID，OrderID是辅助维度，但是由于OrderID其中的6位和UserID完全一致，也就是说，在OrderID中会把UserID打进去。

在路由的时候，如果SQL中带有UserID，那么直接拿UserID进行Hash取模路由；如果SQL中带有的OrderID维度，那么取出OrderID中的6位UserID进行Hash取模路由，结果是一致的。

**适用场景**：辅助维度和主维度其实可以通过将主维度和辅助维度的值进行信息共享

**即基因法：**

分库基因：假如通过uid分库，分为8个库，采用uid%8的方式进行路由，此时是由uid的最后3bit来决定这行User数据具体落到哪个库上，那么这3bit可以看为分库基因。

上面的映射关系的方法需要额外存储映射表，按非uid字段查询时，还需要多一次数据库或cache的访问。如果想要消除多余的存储和查询，可以通过f函数取login_name的基因作为uid的分库基因。生成uid时，参考上文所述的分布式唯一ID生成方案，再加上最后3位bit值=f(login_name)。当查询login_name时，只需计算f(login_name)%8的值，就可以定位到具体的库。不过这样需要提前做好容量规划，预估未来几年的数据量需要分多少库，要预留一定bit的分库基因。


**4）建立索引表**

对于辅助维度可以建一张辅助维度和主维度的映射表。

举例来说，表A有两个维度，主维度a，辅助维度b，目前只有主维度的一份数据。那么建一张新表B_A_Index，里面就只有两个字段，a和b的值，这张表可以分表，也可以不分表，建议分表这张表的主维度就是b。这类kv格式的索引结构，可以很好的使用cache来优化查询性能，而且映射关系不会频繁变更，缓存命中率会很高。

例如：login_name不能直接定位到数据库，可以建立login_name→uid的映射关系，用索引表或缓存来存储。当访问login_name时，先通过映射表查询出login_name对应的uid，再通过uid定位到具体的库。

**试用场景**：主副维度是一一对应的。优势是，无需数据冗余，只需要冗余一份索引数据。缺点是，需要业务进行略微的改造。

### 3、前台与后台分离

对于用户侧，主要需求是以单行查询为主，需要建立login_name/phone/email到uid的映射关系，可以解决这些字段的查询问题。

而对于运营侧，很多批量分页且条件多样的查询，这类查询计算量大，返回数据量大，对数据库的性能消耗较高。此时，如果和用户侧公用同一批服务或数据库，可能因为后台的少量请求，占用大量数据库资源，而导致用户侧访问性能降低或超时。

这类业务最好采用"前台与后台分离"的方案，运营侧后台业务抽取独立的service和db，解决和前台业务系统的耦合。由于运营侧对可用性、一致性的要求不高，可以不访问实时库，而是通过binlog异步同步数据到运营库进行访问。在数据量很大的情况下，还可以使用ES搜索引擎或Hive来满足后台复杂的查询方式。

## 二、分片数的选择

一般水平拆分有两个层次：分库和分表。

### 1、表数目决策

一般情况下，建议单个物理分表的容量不超过1000万行数据。通常可以预估2到5年的数据增长量，用估算出的总数据量除以总的物理分库数，再除以建议的最大数据量1000万，即可得出每个物理分库上需要创建的物理分表数：

（未来3到5年内总共的记录行数）/ 单张表建议记录行数　　（单张表建议记录行数 = 1000万）

表的数量不宜过多，涉及到聚合查询或者分表键在多个表上的SQL语句，就会并发到更多的表上进行查询。举个例子，分了4个表和分了2个表两种情况，一种需要并发到4表上执行，一种只需要并发到2张表上执行，显然后者效率更高。

表的数目不宜过少，少的坏处在于一旦容量不够就又要扩容了，而分库分表的库想要扩容是比较麻烦的。一般建议一次分够。建议表的数目是2的幂次个数，方便未来可能的迁移。

### 2、库数目决策

计算公式：按照存储容量来计算 =（3到5年内的存储容量）/ 单个库建议存储容量　　（单个库建议存储容量 <300G以内）

DBA的操作，一般情况下，会把若干个分库放到一台实例上去。未来一旦容量不够，要发生迁移，通常是对数据库进行迁移。**所以库的数目才是最终决定容量大小。**

最差情况，所有的分库都共享数据库机器。最优情况，每个分库都独占一台数据库机器。一般建议一个数据库机器上存放8个数据库分库。

## 三、分表策略的选择

| 分表方式 | 解释 | 优点 | 缺点 |
| --- | --- | --- | --- |
| **Hash** | 拿分表键的值Hash取模进行路由。最常用的分表方式。 | 数据量散列均衡，每个表的数据量大致相同。<br>请求压力散列均衡，不存在访问热点 | 一旦现有的表数据量需要再次扩容时，需要涉及到数据移动，比较麻烦。所以一般建议是一次性分够。 |
| **Range** | 拿分表键按照ID范围进行路由，比如id在1-10000的在第一个表中，10001-20000的在第二个表中，依次类推。这种情况下，分表键只能是数值类型。 | 数据量可控，可以均衡，也可以不均衡<br>扩容比较方便，因为如果ID范围不够了，只需要调整规则，然后建好新表即可。 | 无法解决热点问题，如果某一段数据访问QPS特别高，就会落到单表上进行操作。 |
| **时间** | 拿分表键按照时间范围进行路由，比如时间在1月的在第一个表中，在2月的在第二个表中，依次类推。这种情况下，分表键只能是时间类型。 | 扩容比较方便，因为如果时间范围不够了，只需要调整规则，然后建好新表即可。 | 数据量不可控，有可能单表数据量特别大，有可能单表数据量特别小<br>无法解决热点问题，如果某一段数据访问QPS特别高，就会落到单表上进行操作。 |