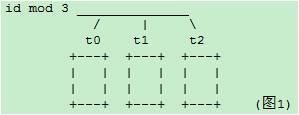
一种可以避免数据迁移的分库分表scale-out扩容方式

目前绝大多数应用采取的两种分库分表规则

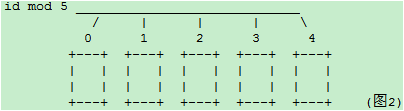
1. mod方式
2. dayofweek系列日期方式（所有星期1的数据在一个库/表,或所有?月份的数据在一个库表）

这两种方式有个本质的特点，就是离散性加周期性。

例如以一个表的主键对3取余数的方式分库或分表：

[](http://rdc.taobao.com/team/jm/files/2010/12/%E5%9B%BE%E7%89%871.png)

那么随着数据量的增大，每个表或库的数据量都是各自增长。当一个表或库的数据量增长到了一个极限，要加库或加表的时候，  
介于这种分库分表算法的离散性，必需要做数据迁移才能完成。例如从3个扩展到5个的时候：

[](http://rdc.taobao.com/team/jm/files/2010/12/%E5%9B%BE%E7%89%872.png)

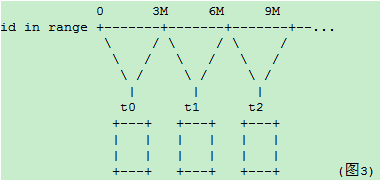
需要将原先以mod3分类的数据，重新以mod5分类，不可避免的带来数据迁移。每个表的数据都要被重新分配到多个新的表  
相似的例子比如从dayofweek分的7个库/表,要扩张为以dayofmonth分的31张库/表，同样需要进行数据迁移。

数据迁移带来的问题是

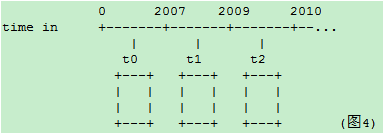
1. 业务至少要两次发布
2. 要专门写工具来导数据。由于各业务之间的差别，很难做出统一的工具。目前几乎都是每个业务写一套
3. 要解决增量、全量、时间点，数据不一致等问题

如何在数据量扩张到现有库表极限，加库加表时避免数据迁移呢？  
通常的数据增长往往是随着时间的推移增长的。随着业务的开展，时间的推移，数据量不断增加。（不随着时间增长的情况，  
例如某天突然需要从另一个系统导入大量数据，这种情况完全可以由dba依据现有的分库分表规则来导入，因此不考虑这种问题。）

考虑到数据增长的特点，如果我们以代表时间增长的字段，按递增的范围分库，则可以避免数据迁移  
例如，如果id是随着时间推移而增长的全局sequence，则可以以id的范围来分库：（全局sequence可以用tddl现在的方式也可以用ZooKeeper实现）  
id在 0–100万在第一个库中，100-200万在第二个中，200-300万在第3个中 （用M代表百万数据）

[](http://rdc.taobao.com/team/jm/files/2010/12/%E5%9B%BE%E7%89%873.png)

或者以时间字段为例，比如一个字段表示记录的创建时间，以此字段的时间段分库gmt\_create\_time in range

[](http://rdc.taobao.com/team/jm/files/2010/12/%E5%9B%BE%E7%89%874.png)

这样的方式下，在数据量再增加达到前几个库/表的上限时，则继续水平增加库表，原先的数据就不需要迁移了  
但是这样的方式会带来一个热点问题：当前的数据量达到某个库表的范围时，所有的插入操作，都集中在这个库/表了。

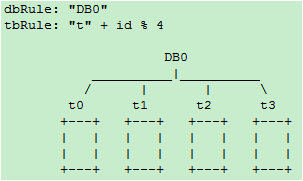
所以在满足基本业务功能的前提下，分库分表方案应该尽量避免的两个问题：

1. 数据迁移  
2. 热点

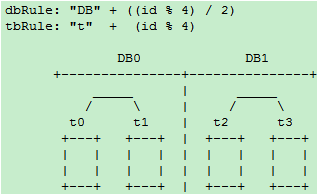
如何既能避免数据迁移又能避免插入更新的热点问题呢？  
结合离散分库/分表和连续分库/分表的优点，如果一定要写热点和新数据均匀分配在每个库，同时又保证易于水平扩展，可以考虑这样的模式：

**【水平扩展scale-out方案模式一】**

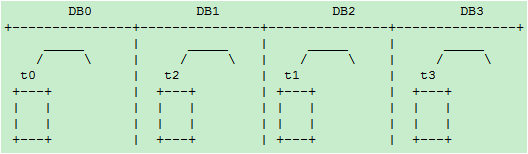
阶段一：一个库DB0之内分4个表，id%4 ：

[](http://rdc.taobao.com/team/jm/files/2010/12/%E5%9B%BE%E7%89%875.png)

阶段二：增加db1库，t2和t3整表搬迁到db1

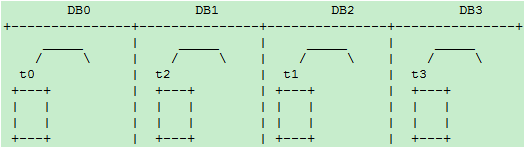
[](http://rdc.taobao.com/team/jm/files/2010/12/%E5%9B%BE%E7%89%876.png)

阶段三：增加DB2和DB3库，t1整表搬迁到DB2，t3整表搬迁的DB3：

[](http://rdc.taobao.com/team/jm/files/2010/12/%E5%9B%BE%E7%89%877.png)

为了规则表达，通过内部名称映射或其他方式，我们将DB1和DB2的名称和位置互换得到下图：

dbRule: “DB” + (id % 4)  
tbRule: “t”  + (id % 4)

[](http://rdc.taobao.com/team/jm/files/2010/12/%E5%9B%BE%E7%89%878.png)

这样3个阶段的扩展方案中，每次次扩容只需要做一次停机发布，不需要做数据迁移。停机发布中只需要做整表搬迁。  
这个相对于每个表中的数据重新分配来说，不管是开发做，还是DBA做都会简单很多。

如果更进一步数据库的设计和部署上能做到每个表一个硬盘，那么扩容的过程只要把原有机器的某一块硬盘拔下来，  
插入到新的机器上，就完成整表搬迁了！可以大大缩短停机时间。

具体在mysql上可以以库为表。开始一个物理机上启动4个数据库实例，每次倍增机器，直接将库搬迁到新的机器上。  
这样从始至终规则都不需要变化，一直都是：

dbRule: “DB” + (id % 4)  
tbRule: “t”  + (id % 4)

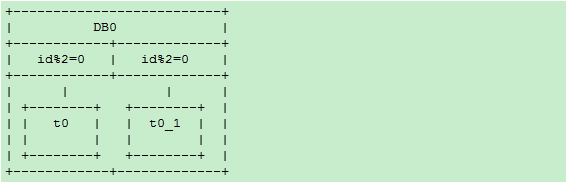
即逻辑上始终保持4库4表，每个表一个库。这种做法也是目前店铺线图片空间采用的做法。

上述方案有一个缺点，就是在从一个库到4个库的过程中，单表的数据量一直在增长。当单表的数据量超过一定范围时，可能会带来性能问题。比如索引的问题，历史数据清理的问题。  
另外当开始预留的表个数用尽，到了4物理库每库1个表的阶段，再进行扩容的话，不可避免的要从表上下手。那么我们来考虑表内数据上限不增长的方案：

**【水平扩展scale-out方案模式二】**

阶段一：一个数据库，两个表，rule0 = id % 2

分库规则dbRule: “DB0″  
分表规则tbRule: “t” + (id % 2)

[](http://rdc.taobao.com/team/jm/files/2010/12/%E5%9B%BE%E7%89%879.png)

阶段二：当单库的数据量接近1千万，单表的数据量接近500万时，进行扩容（数据量只是举例，具体扩容量要根据数据库和实际压力状况决定）：  
增加一个数据库DB1，将DB0.t1整表迁移到新库DB1。  
每个库各增加1个表，未来10M-20M的数据mod2分别写入这2个表：t0\_1，t1\_1：

分库规则dbRule:

“DB” + (id % 2)

分表规则tbRule:

if(id < 1千万){

return "t"+ (id % 2); //1千万之前的数据，仍然放在t0和t1表。t1表从DB0搬迁到DB1库

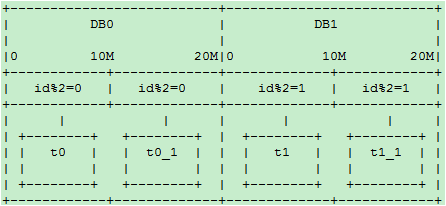
}else if(id < 2千万){

return "t"+ (id % 2) +"\_1"; //1千万之后的数据，各放到两个库的两个表中: t0\_1,t1\_1

}else{

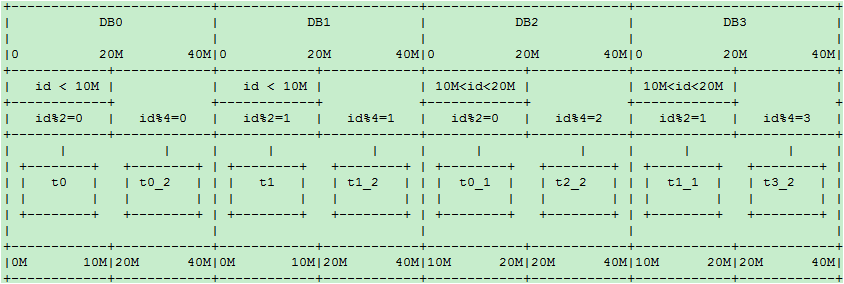
throw new IllegalArgumentException("id outof range[20000000]:" + id);

}

[](http://rdc.taobao.com/team/jm/files/2010/12/%E5%9B%BE%E7%89%8710.png)

这样10M以后的新生数据会均匀分布在DB0和DB1; 插入更新和查询热点仍然能够在每个库中均匀分布。  
每个库中同时有老数据和不断增长的新数据。每表的数据仍然控制在500万以下。

阶段三：当两个库的容量接近上限继续水平扩展时，进行如下操作：  
新增加两个库：DB2和DB3. 以id % 4分库。余数0、1、2、3分别对应DB的下标. t0和t1不变，  
将DB0.t0\_1整表迁移到DB2; 将DB1.t1\_1整表迁移到DB3  
20M-40M的数据mod4分为4个表：t0\_2，t1\_2，t2\_2，t3\_2，分别放到4个库中：

[](http://rdc.taobao.com/team/jm/files/2010/12/%E5%9B%BE%E7%89%8711.png)

新的分库分表规则如下：

分库规则dbRule:

if(id < 2千万){

//2千万之前的数据，4个表分别放到4个库

if(id < 1千万){

return "db"+ (id % 2); //原t0表仍在db0, t1表仍在db1

}else{

return "db"+ ((id % 2) +2); //原t0\_1表从db0搬迁到db2; t1\_1表从db1搬迁到db3

}

}else if(id < 4千万){

return "db"+ (id % 4); //超过2千万的数据，平均分到4个库

}else{

throw new IllegalArgumentException("id out of range. id:"+id);

}

分表规则tbRule:

if(id < 2千万){ //2千万之前的数据，表规则和原先完全一样，参见阶段二

if(id < 1千万){

return "t"+ (id % 2); //1千万之前的数据，仍然放在t0和t1表

}else{

return "t"+ (id % 2) +"\_1"; //1千万之后的数据，仍然放在t0\_1和t1\_1表

}

}else if(id < 4千万){

return "t"+ (id % 4)+"\_2"; //超过2千万的数据分为4个表t0\_2，t1\_2，t2\_2，t3\_2

}else{

throw new IllegalArgumentException("id out of range. id:"+id);

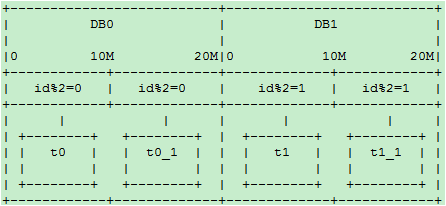
}

随着时间的推移，当第一阶段的t0/t1，第二阶段的t0\_1/t1\_1逐渐成为历史数据，不再使用时，可以直接truncate掉整个表。省去了历史数据迁移的麻烦。

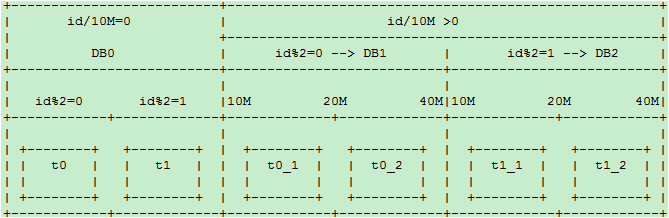
上述3个阶段的分库分表规则在TDDL2.x中已经全部支持，具体请咨询TDDL团队。

**【水平扩展scale-out方案模式三】**

非倍数扩展：如果从上文的阶段二到阶段三不希望一下增加两个库呢？尝试如下方案：

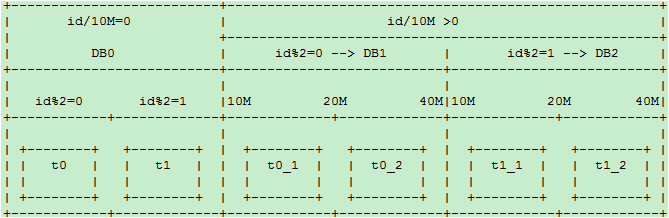
迁移前：  
[](http://rdc.taobao.com/team/jm/files/2010/12/%E5%9B%BE%E7%89%8710.png)

新增库为DB2，t0、t1都放在DB0，  
t0\_1整表迁移到DB1  
t1\_1整表迁移到DB2

迁移后：  
[](http://rdc.taobao.com/team/jm/files/2010/12/%E5%9B%BE%E7%89%8712.png)

这时DB0退化为旧数据的读库和更新库。新增数据的热点均匀分布在DB1和DB2  
4无法整除3，因此如果从4表2库扩展到3个库，不做行级别的迁移而又保证热点均匀分布看似无法完成。

当然如果不限制每库只有两个表，也可以如下实现：

[](http://rdc.taobao.com/team/jm/files/2010/12/%E5%9B%BE%E7%89%87121.png)

小于10M的t0和t1都放到DB0，以mod2分为两个表，原数据不变  
10M-20M的，以mod2分为两个表t0\_1、t1\_1，原数据不变，分别搬迁到DB1，和DB2  
20M以上的以mod3平均分配到3个DB库的t\_0、t\_2、t\_3表中  
这样DB1包含最老的两个表，和最新的1/3数据。DB1和DB2都分表包含次新的两个旧表t0\_1、t1\_1和最新的1/3数据。  
新旧数据读写都可达到均匀分布。

总而言之：  
两种规则映射（函数）：

1. 离散映射：如mod或dayofweek， 这种类型的映射能够很好的解决热点问题，但带来了数据迁移和历史数据问题。
2. 连续映射；如按id或gmt\_create\_time的连续范围做映射。这种类型的映射可以避免数据迁移，但又带来热点问题。

离散映射和连续映射这两种相辅相成的映射规则，正好解决热点和迁移这一对相互矛盾的问题。  
我们之前只运用了离散映射，引入连续映射规则后，两者结合，精心设计，  
应该可以设计出满足避免热点和减少迁移之间任意权衡取舍的规则。

基于以上考量，分库分表规则的设计和配置，长远说来必须满足以下要求

1. 可以动态推送修改
2. 规则可以分层级叠加，旧规则可以在新规则下继续使用，新规则是旧规则在更宽尺度上的拓展，以此支持新旧规则的兼容，避免数据迁移
3. 用mod方式时，最好选2的指数级倍分库分表，这样方便以后切割。