# 背景

数据库数据会随着业务的发展而不断增多，因此数据操作，如增删改查的开销也会越来越大。

再加上物理服务器的资源有限（CPU、磁盘、内存、IO 等）。最终数据库所能承载的数据量、数据处理能力都将遭遇瓶颈。

换句话说需要合理的数据库架构来存放不断增长的数据，这个就是分库分表的设计初衷。目的就是为了缓解数据库的压力，最大限度提高数据操作的效率。

当一张表的数据达到几千万时，你查询一次所花的时间会变多，如果有联合查询的话，很有可能会“卡死”。分表的目的就在于此，减小数据库的负担，缩短查询时间。

MySQL中有一种机制是表锁定和行锁定，是为了保证数据的完整性。表锁定表示你们都不能对这张表进行操作，必须等我对表操作完才行。行锁定也一样，别的SQL必须等待对这条数据操作完了，才能对这条数据进行操作。

MySQL的主从复制解决了数据库的读写分离，并很好的提升了读的性能。但是，主从复制也带来其他一系列性能瓶颈问题：

1、写入无法扩展

2、写入无法缓存

**3、复制延时**

**4、锁表率上升**

**5、表变大，缓存率下降**

那问题产生总得解决的，这就产生下面的分区分表优化方案。

不管是IO瓶颈，还是CPU瓶颈，最终都会导致数据库的活跃连接数增加，进而逼近甚至达到数据库可承载活跃连接数的阈值。在业务Service来看就是，可用数据库连接少甚至无连接可用（并发量、吞吐量、崩溃）。

## IO瓶颈

第一种：磁盘读IO瓶颈，热点数据太多，数据库缓存放不下，每次查询时会产生大量的IO，降低查询速度->分库和垂直分表。

第二种：网络IO瓶颈，请求的数据太多，网络带宽不够->分库。

## CPU瓶颈

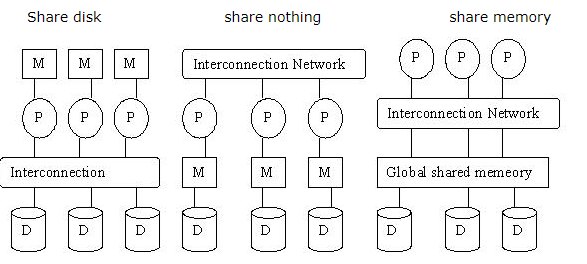
第一种：SQL问题，如SQL中包含join，group by，order by，非索引字段条件查询等，增加CPU运算的操作 -> SQL优化，建立合适的索引，在业务Service层进行业务计算。

第二种：单表数据量太大，查询时扫描的行太多，SQL效率低，CPU率先出现瓶颈 -> 水平分表。

# 概述

Sharding的基本思想就要把一个数据库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。

数据库构架设计中主要有Shared Everthting、Shared Nothing、和Shared Disk：



## Shared Everthting

Shared Everthting:一般是针对单个主机，完全透明共享CPU/MEMORY/IO，并行处理能力是最差的，典型的代表SQLServer。

注：即所有的组件都是共享的。

## Shared Disk

Shared Disk：各个处理单元使用自己的私有CPU和Memory，共享磁盘系统。典型的代表Oracle Rac，它是数据共享，可通过增加节点来提高并行处理的能力，扩展能力较好。其类似于SMP（对称多处理）模式，但是当存储器接口达到饱和的时候，增加节点并不能获得更高的性能。

## Shared Nothing

**无共享架构**，有时也称为水平扩展（horizontal scale）或向外扩展（scale out）。

无共享架构的各个节点之间的通信都是软件层面使用网络实现，不同产品在架构不同导致这个细节也不同。有些架构是计算与存储分离。计算节点特点是无状态（即数据不要求持久化），通过集群方式管理，可以水平扩展；存储节点有数据，使用复制和分区技术，节点间任务集中调度或者独立交互。

Shared Nothing：各个处理单元都有自己私有的CPU/内存/硬盘等，不存在共享资源，类似于MPP（大规模并行处理）模式，各处理单元之间通过协议通信，并行处理和扩展能力更好。典型代表DB2 DPF和Hadoop，各节点相互独立，各自处理自己的数据，处理后的结果可能向上层汇总或在节点间流转。

我们常说的 Sharding 其实就是Share Nothing架构，它是把某个表从物理存储上被水平分割，并分配给多台服务器（或多个实例），每台服务器可以独立工作，具备共同的schema，比如MySQL Proxy和Google的各种架构，只需增加服务器数就可以增加处理能力和容量。

Shared nothing架构（shared nothing architecture）是一种分布式计算架构。这种架构中的每一个节点（node）都是独立、自给的，而且整个系统中没有单点竞争。

在一个纯Shared Nothing系统中，通过简单地增加一些廉价的计算机做为系统的节点却可以获取几乎无限的扩展。

Shared nothing系统通常需要将他的数据分布在多个节点的不同数据库中（不同的计算机处理不同的用户和查询）或者要求每个节点通过使用某些协调协议来保留它自己的应用程序数据备份 ，这通常被成为数据库Sharding。

# 原则

## 尽量减少事务边界

所谓的事务边界即是指单个SQL语句在后端数据库上同时执行的数量，事务边界大的典型示例，即一条SQL语句同时被推送到后端所有数据库中运行。事务边界的数量越大，会给系统带来以下弊端：

系统的锁冲突概率越高。如果事务边界大的SQL请求比较多，在一次SQL请求处理过程中自然对于后端的数据库操作的数据库记录覆盖比较广，当有多个类似的SQL请求并行执行时，则出现数据锁造成的资源访问互斥的概率会大大增加。

系统越难以扩展。如果有大量的SQL请求都是这样全表扫描，或者从极端角度说明这个问题，如果每一次的SQL请求都需要全表扫描执行，你会发现整个平台的数据库连接数量是取决于后端单个数据库的连接能力，也就意味着整个数据库的能力是无法通过增加后端数据库实例来扩展的。所以如果有大量的全表扫描的SQL请求对于系统的扩展能力会带来不小的影响。

整体性能越低。对于性能，这里想强调的是对系统整体性能的影响，而不是单次SQL的性能。

## 异构索引表尽量降低全表扫描频率

基于订单数据的分库分表场景，按照订单ID取模虽然很好地满足了订单数据均匀地保存在后端数据库中，但在买家查看自己订单的业务场景中，就出现了全表扫描的情况，而且买家查看自己订单的请求是非常频繁的，必然给数据库带来扩展或性能的问题，有违“尽量减少事务边界”这一原则。其实这类场景还有很多，比如卖家要查看与自己店铺相关的订单信息，同样也会出现上述所说的大量进行全表扫描的SQL请求。

针对这类场景问题，最常用的是采用“异构索引表”的方式解决，即采用异步机制将原表内的每一次创建或更新，都换另一个维度保存一份完整的数据表或索引表。本质上这是互联网公司很多时候都采用的一个解决思路：“拿空间换时间”。

也就是应用在创建或更新一条按照订单ID为分库分表键的订单数据时，也会再保存一份按照买家ID为分库分表键的订单索引数据，其结果就是同一买家的所有订单索引表都保存在同一数据库中，这就是给订单创建了异构索引表。

# 分库

## 概述

分库，是指将一个应用的数据库分拆为多个数据库。例如，一个新闻网站，原始状态，用户表、新闻表、评论表都在同一个数据库。后期，将用户表数据放到单独的用户数据库中，将评论表的数据放到单独的评论数据库中。

**分表解决的是数据量多大的问题，分库解决的是数据库性能瓶颈问题**。

单纯的分表可以解决数据量过大导致索引变慢的问题，但是无法解决多并发请求访问同一个库导致数据库响应变慢的问题，所以通常水平拆分都至少要采用分库的方式（垂直拆分采用分表的方式），用于一并解决大数据量和高并发的问题。

## 分类

### 单库单表

单库单表是最常见的数据库设计，例如，有一张用户(user)表放在数据库db中，所有的用户都可以在db库中的user表中查到。

### 单库多表

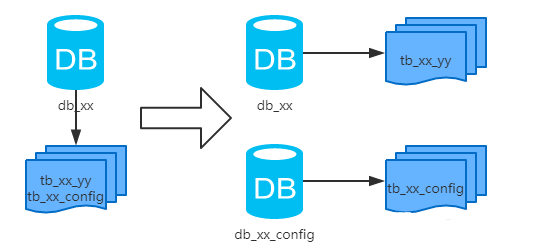
随着用户数量的增加，user表的数据量会越来越大，当数据量达到一定程度的时候对user表的查询会渐渐的变慢，从而影响整个DB的性能。如果使用mysql,还有一个更严重的问题是，当需要添加一列的时候，mysql会锁表，期间所有的读写操作只能等待。

可以通过某种方式将user进行水平的切分，产生两个表结构完全一样的user\_0000,user\_0001等表，user\_0000+user\_0001+…的数据刚好是一份完整的数据。

### 多库多表

随着数据量增加也许单台DB的存储空间不够，随着查询量的增加单台数据库服务器已经没办法支撑。这个时候可以再对数据库进行水平区分。

## 垂直拆分



**概念：**以表为依据，按照业务归属不同，将不同的表拆分到不同的库中。

**结果：**

每个库的结构都不一样；

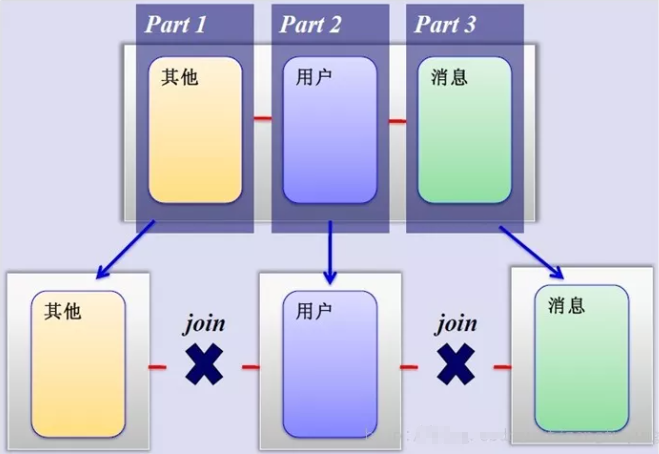
每个库的数据也不一样，没有交集；

所有库的并集是全量数据；

**场景：**系统绝对并发量上来了，并且可以抽象出单独的业务模块。

**分析：**到这一步，基本上就可以服务化了。例如，随着业务的发展一些公用的配置表、字典表等越来越多，这时可以将这些表拆到单独的库中，甚至可以服务化。再有，随着业务的发展孵化出了一套业务模式，这时可以将相关的表拆到单独的库中，甚至可以服务化。

如果把业务切割得足够独立，那把不同业务的数据放到不同的数据库服务器将是一个不错的方案，而且万一其中一个业务崩溃了也不会影响其他业务的正常进行，并且也起到了负载分流的作用，大大提升了数据库的吞吐能力。经过垂直分区后的数据库架构图如下：



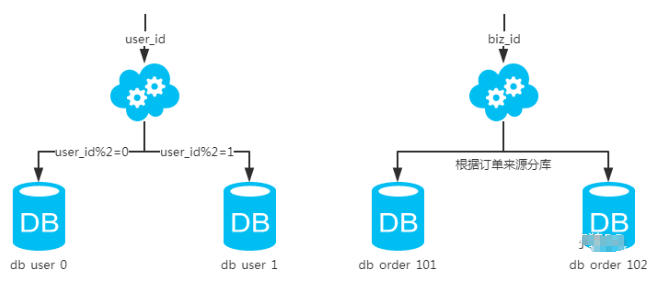
然而，尽管业务之间已经足够独立了，但是有些业务之间或多或少总会有点联系，如用户，基本上都会和每个业务相关联，况且这种分区方式，也不能解决单张表数据量暴涨的问题，因此为何不试试水平分割呢？

将系统中不存在关联关系或者需要join的表可以放在不同的数据库不同的服务器中。

按照业务垂直划分，比如可以按照业务分为资金、会员、订单三个数据库。

需要解决的问题：跨数据库的事务、join查询等问题。

## 水平拆分



**概念：**以字段为依据，按照一定策略（hash、range等），将一个库中的数据拆分到多个库中。

**结果：**

每个库的结构都一样；

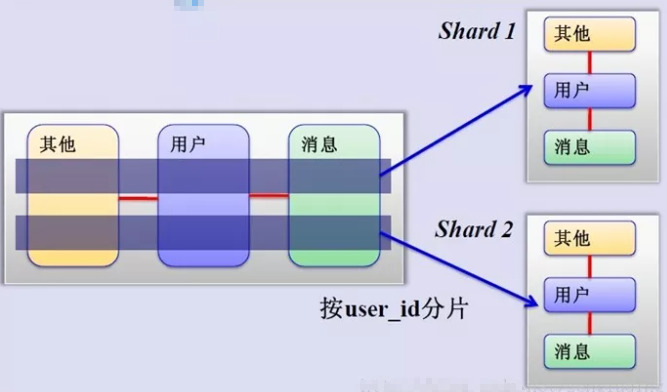
**每个库的数据都不一样，没有交集**；

所有库的并集是全量数据；

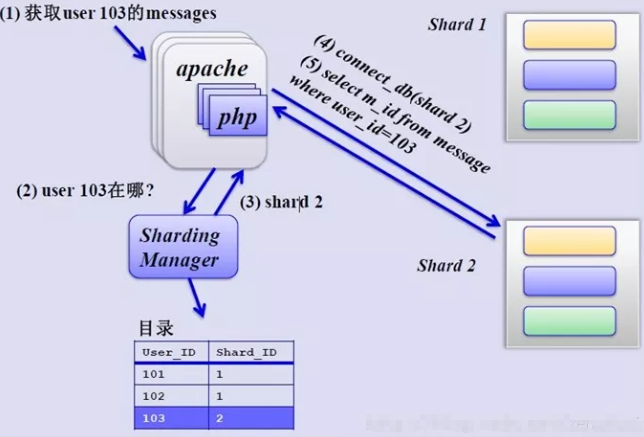
**场景：**系统绝对并发量上来了，分表难以根本上解决问题，并且还没有明显的业务归属来垂直分库。

**分析：**库多了，IO和CPU的压力自然可以成倍缓解。

这是一个非常好的思路，将用户按一定规则（按id哈希）分组，并把该组用户的数据存储到一个数据库分片中，即一个sharding，这样随着用户数量的增加，只要简单地配置一台服务器即可，原理图如下：



如何来确定某个用户所在的shard？可以建一张用户和shard对应的数据表，每次请求先从这张表找用户的shardid，再从对应shard中查询相关数据，如下图所示：



按照规则划分，一般水平分库是在垂直分库之后的。比如每天处理的订单数量是海量的，可以按照一定的规则水平划分。比如大部分的站点，数据都是和用户相关，那么可以根据用户，将数据按照用户水平拆分。

需要解决的问题：数据路由、组装。

## 作用

其主要目的是为突破单节点数据库服务器I/O能力限制，解决数据库拓展性问题。

## 应用

什么时候考虑使用分库？

1. 单台DB的存储空间不够；
2. 随着查询量的增加单台数据库服务器已经没办法支撑。

但分表也有不可替代的适用场景。最常见的分表需求是事务问题，同在一个库则不需要考虑考虑分布式事务，善于使用同库不同表可有效避免分布式事务带来的麻烦。目前强一致性的分布式事务由于性能问题，导致使用起来并不一定比不分库分表快。目前采用最终一致性的柔性事务居多。分表的另一个存在的理由是，过多的数据库实力不利于运维管理。

综上所述，最佳实践是合理地配合使用分库+分表。

# 分表

## 概述

关系型数据库在大于一定数据量的情况下检索性能会急剧下降。在面对互联网海量数据情况时，所有数据都存于一张表，显然会轻易超过数据库可承受的数据量阈值。这个单表可承受的数据量阈值，需根据数据库和并发量的差异，通过实际测试获得。

当一张表的数据达到几千万时，你查询一次所花的时间会变多，如果有联合查询的话，我想有可能会死在那儿了。分表的目的就在于此，减小数据库的负担，缩短查询时间。

MySQL中有一种机制是表锁定和行锁定，是为了保证数据的完整性。表锁定表示你们都不能对这张表进行操作，必须等我对表操作完才行。行锁定也一样，别的SQL必须等我对这条数据操作完了，才能对这条数据进行操作。

基于数据查询和锁的考虑，引入分表。

分表，从表面意思上看呢，就是把一张表分成N多个小表。例如，将用户表user的数据分拆到活跃用户表user\_active和非活跃用户表user\_inaction。

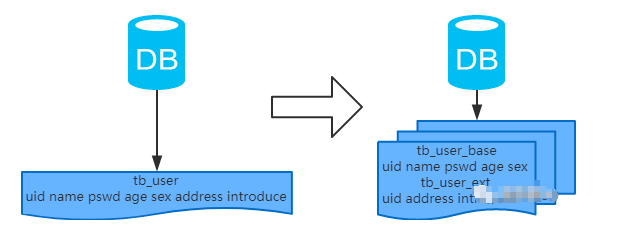
水平拆分如果能预估规模，越早造成本越低。

分区就是把一张表的数据分成N个区块，在逻辑上看最终只是一张表，但底层是由N个物理区块组成的。

分表就是把一张表按照一定的规则分解成N个具有独立存储空间的实体表。系统读写时需要根据定义好的规则得到对应的字表名，然后操作它。

## 分类

### 垂直分表



**概念：**以字段为依据，按照字段的活跃性，将表中字段拆到不同的表（主表和扩展表）中。

**结果：**

每个表的结构都不一样；

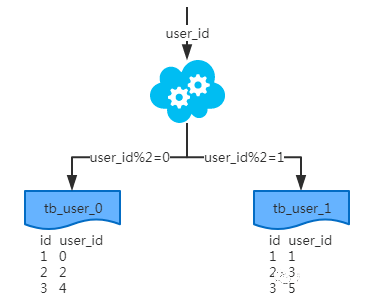
每个表的数据也不一样，一般来说，每个表的字段至少有一列交集，一般是主键，用于关联数据；

所有表的并集是全量数据；

**场景：**系统绝对并发量并没有上来，表的记录并不多，但是字段多，并且热点数据和非热点数据在一起，单行数据所需的存储空间较大。以至于数据库缓存的数据行减少，查询时会去读磁盘数据产生大量的随机读IO，产生IO瓶颈。

**分析：**可以用列表页和详情页来帮助理解。垂直分表的拆分原则是将热点数据（可能会冗余经常一起查询的数据）放在一起作为主表，非热点数据放在一起作为扩展表。这样更多的热点数据就能被缓存下来，进而减少了随机读IO。拆了之后，要想获得全部数据就需要关联两个表来取数据。但记住，千万别用join，因为join不仅会增加CPU负担并且会讲两个表耦合在一起（必须在一个数据库实例上）。关联数据，应该在业务Service层做文章，分别获取主表和扩展表数据然后用关联字段关联得到全部数据。

### 水平分表



**概念：**以字段为依据，按照一定策略（hash、range等），将一个表中的数据拆分到多个表中。

**结果：**

每个表的结构都一样；

每个表的数据都不一样，没有交集；

所有表的并集是全量数据；

**场景：**系统绝对并发量并没有上来，只是单表的数据量太多，影响了SQL效率，加重了CPU负担，以至于成为瓶颈。

**分析：**表的数据量少了，单次SQL执行效率高，自然减轻了CPU的负担。

## 算法

### Hash分片

参考：<http://mysql.taobao.org/monthly/2022/06/05/>

### Range分片

### LIST分片

### 广播表

## 作用

分表后，单表的并发能力提高了，磁盘I/O性能也提高了，写操作效率提高了：

1. 查询一次的时间短了；
2. 数据分布在不同的文件，磁盘I/O性能提高；
3. 读写锁影响的数据量小；
4. 插入数据库需要重新建立索引的数据减少。

## 应用

什么时候考虑分表？

1. 一张表的查询速度已经慢到影响使用的时候；
2. sql经过优化；
3. 数据量大；
4. 当频繁插入或者联合查询时，速度变慢。

# 对比

## 区别

### 分库+分表

分库分表区别：

1、什么是分库分表？

从字面上简单理解，分库就是把原本存储于一个库的数据分块存储到多个库上，分表就是把原本存储于一个表的数据分块存储到多个表上。

2、为什么要分库分表？

数据库中的数据量不一定是可控的，在未进行分库分表的情况下，随着时间和业务的发展，库中的表会越来越多，表中的数据量也会越来越大，相应地，数据操作，增删改查的开销也会越来越大；另外，一台服务器的资源（CPU、磁盘、内存、IO等）是有限的，最终数据库所能承载的数据量、数据处理能力都将遭遇瓶颈,。

3、分库分表的实施策略

如果你的单机性能很低了，那可以尝试分库。分库，业务透明，在物理实现上分成多个服务器，不同的分库在不同服务器上。分区可以把表分到不同的硬盘上，但不能分配到不同服务器上。一台机器的性能是有限制的，用分库可以解决单台服务器性能不够，或者成本过高问题。

当分区之后，表还是很大，处理不过来，这时候可以用分库。

orderid,userid,ordertime,.....

userid%4=0，用分库1

userid%4=1，用分库2

userid%4=2, 用分库3

userid%4=3，用分库4

上面这个就是一个简单的分库路由，根据userid选择分库，即不同的服务器

4、分库分表存在的问题

4.1 事务问题

在执行分库分表之后，由于数据存储到了不同的库上，数据库事务管理出现了困难。如果依赖数据库本身的分布式事务管理功能去执行事务，将付出高昂的性能代价；如果由应用程序去协助控制，形成程序逻辑上的事务，又会造成编程方面的负担。

4.2 跨库跨表的join问题

在执行了分库分表之后，难以避免会将原本逻辑关联性很强的数据划分到不同的表、不同的库上，这时，表的关联操作将受到限制，我们无法join位于不同分库的表，也无法join分表粒度不同的表，结果原本一次查询能够完成的业务，可能需要多次查询才能完成。

4.3 额外的数据管理负担和数据运算压力。

额外的数据管理负担，最显而易见的就是数据的定位问题和数据的增删改查的重复执行问题，这些都可以通过应用程序解决，但必然引起额外的逻辑运算，例如，对于一个记录用户成绩的用户数据表userTable，业务要求查出成绩最好的100位，在进行分表之前，只需一个order by语句就可以搞定，但是在进行分表之后，将需要n个order by语句，分别查出每一个分表的前100名用户数据，然后再对这些数据进行合并计算，才能得出结果。

### 分表+分区

1、实现方式上

a) mysql的分表是真正的分表，一张表分成很多表后，每一个小表都是完正的一张表，都对应三个文件，一个.MYD数据文件，.MYI索引文件，.frm表结构文件。

SQL代码

[root@BlackGhost test]# ls |grep user

alluser.MRG

alluser.frm

user1.MYD

user1.MYI

user1.frm

user2.MYD

user2.MYI

user2.frm

简单说明一下，上面的分表呢是利用了merge存储引擎（分表的一种），alluser是总表，下面有二个分表，user1，user2。他们二个都是独立 的表，取数据的时候，我们可以通过总表来取。这里总表是没有.MYD,.MYI这二个文件的，也就是说，总表他不是一张表，没有数据，数据都放在分表里面。我们来看看.MRG到底是什么东西

Sql代码

[root@BlackGhost test]# cat alluser.MRG |more

user1

user2

#INSERT\_METHOD=LAST

从上面我们可以看出，alluser.MRG里面就存了一些分表的关系，以及插入数据的方式。可以把总表理解成一个外壳，或者是联接池。

b) 分区不一样，一张大表进行分区后，他还是一张表，不会变成二张表，但是他存放数据的区块变多了。

[root@BlackGhost test]# ls |grep aa

aa#P#p1.MYD

aa#P#p1.MYI

aa#P#p3.MYD

aa#P#p3.MYI

aa.frm

aa.par

从上面我们可以看出，aa这张表，分为二个区，p1和p3，本来是三个区，被我删了一个区。我们都知道一张表对应三个文件.MYD,.MYI,.frm。分 区呢根据一定的规则把数据文件和索引文件进行了分割，还多出了一个.par文件，打开.par文件后你可以看出他记录了，这张表的分区信息，根分表中 的.MRG有点像。分区后，还是一张，而不是多张表。

如:

orderid,userid,ordertime,.....

ordertime<2015-01-01 #p0

ordertime<2015-04-01 #p1

ordertime<2015-07-01 #p2

ordertime<2015-10-01 #p3

ordertime<2016-01-01 #p4

按照时间分区。大部分只查询最近的订单数据，那么大部分只访问一个分区，比整个表小多了，数据库可以更加好的缓存，性能也提高了。这个是数据库分的，应用程序透明，无需修改。

2、数据处理上

a) 分表后，数据都是存放在分表里，总表只是一个外壳，存取数据发生在一个一个的分表里面。看下面的例子：

select \* from alluser where id='12'表面上看，是对表alluser进行操作的，其实不是的。是对alluser里面的分表进行了操作。

b) 分区，不存在分表的概念，分区只不过把存放数据的文件分成了许多小块，分区后的表呢，还是一张表。数据处理还是由自己来完成。

3、提高性能上

a) 分表后，单表的并发能力提高了，磁盘I/O性能也提高了。并发能力为什么提高了呢，因为查寻一次所花的时间变短了，如果出现高并发的话，总表可以根据不同的查询，将并发压力分到不同的小表里面。磁盘I/O性能怎么搞高了，本来一个非常大的.MYD文件现在也分摊到各个小表的.MYD中去了。

b) mysql提出了分区的概念，我觉得就想突破磁盘I/O瓶颈，想提高磁盘的读写能力，来增加mysql性能。

在这一点上，分区和分表的测重点不同，分表重点是存取数据时，如何提高mysql并发能力上；而分区呢，如何突破磁盘的读写能力，从而达到提高mysql性能的目的。

4)、实现的难易度上

a) 分表的方法有很多，用merge来分表，是最简单的一种方式。这种方式根分区难易度差不多，并且对程序代码来说可以做到透明的。如果是用其他分表方式就比分区麻烦了。

b) 分区实现是比较简单的，建立分区表，根建平常的表没什么区别，并且对开代码端来说是透明的。

## 联系

分表+分区

1、都能提高mysql的性高，在高并发状态下都有一个良好的表面。

2、分表和分区不矛盾，可以相互配合的，对于那些大访问量，并且表数据比较多的表，我们可以采取分表和分区结合的方式（如果merge这种分表方式，不能和分区配合的话，可以用其他的分表试），访问量不大，但是表数据很多的表，我们可以采取分区的方式等。

# 分库分表步骤

根据容量（当前容量和增长量）评估分库或分表个数 -> 选key（均匀）-> 分表规则（hash或range等）-> 执行（一般双写）-> 扩容问题（尽量减少数据的移动）。

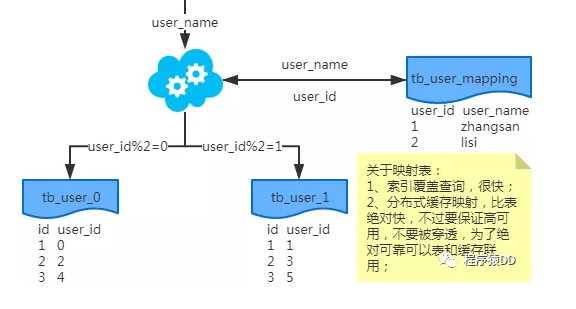
# 分库分表问题

## 非partition key的查询问题

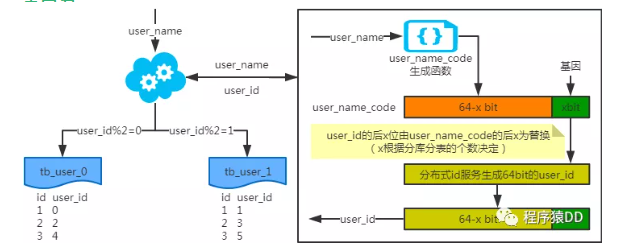
基于水平分库分表，拆分策略为常用的hash法。

1、端上除了partition key只有一个非partition key作为条件查询

映射法



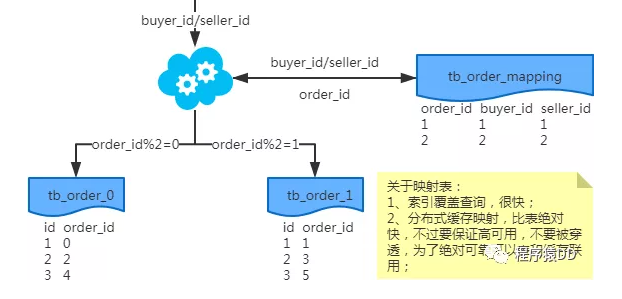
基因法



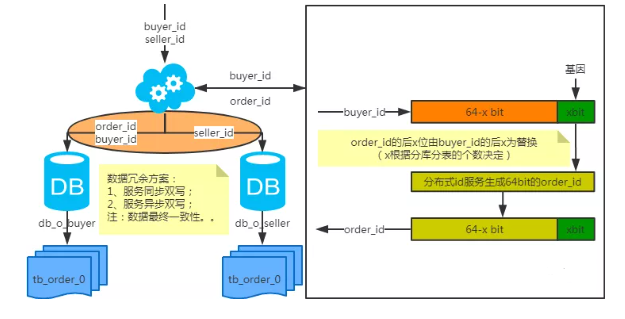
注：写入时，基因法生成user\_id，如图。关于xbit基因，例如要分8张表，23=8，故x取3，即3bit基因。根据user\_id查询时可直接取模路由到对应的分库或分表。根据user\_name查询时，先通过user\_name\_code生成函数生成user\_name\_code再对其取模路由到对应的分库或分表。id生成常用snowflake算法。

2、端上除了partition key不止一个非partition key作为条件查询

映射法



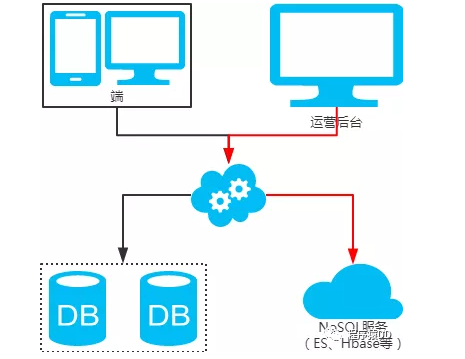
冗余法



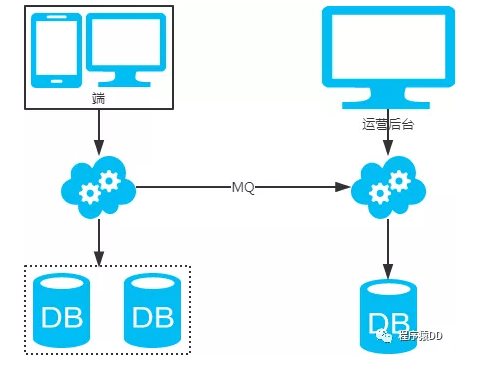
注：按照order\_id或buyer\_id查询时路由到db\_o\_buyer库中，按照seller\_id查询时路由到db\_o\_seller库中。感觉有点本末倒置！有其他好的办法吗？改变技术栈呢？

3、后台除了partition key还有各种非partition key组合条件查询

NoSQL法



冗余法



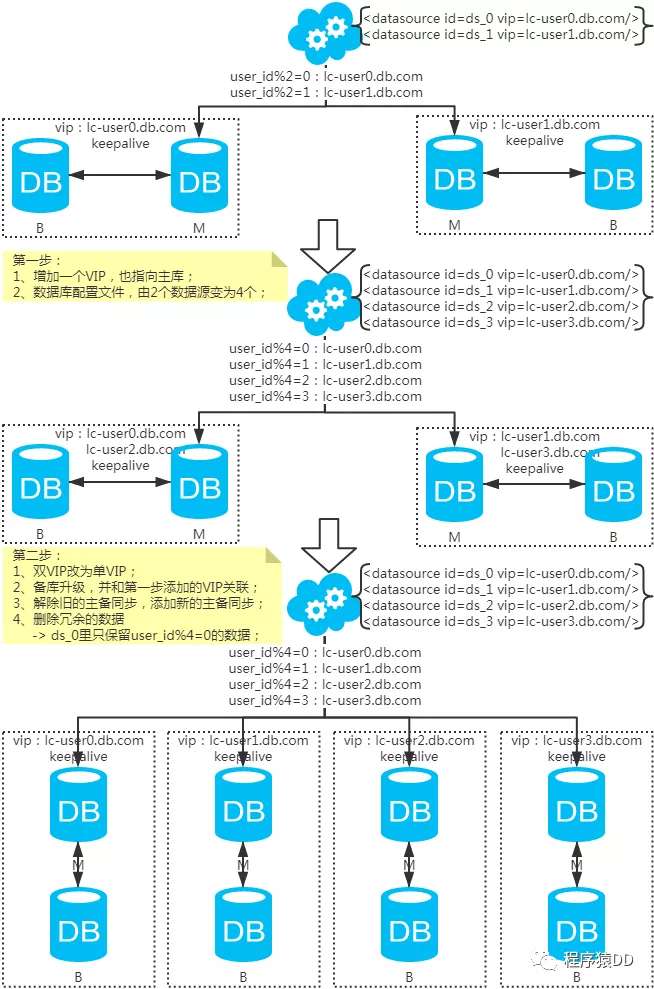
## 非partition key跨库跨表分页查询问题

基于水平分库分表，拆分策略为常用的hash法。

## 扩容问题

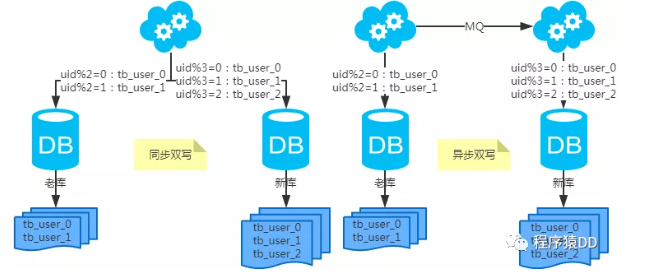
基于水平分库分表，拆分策略为常用的hash法。

1. 水平扩容库（升级从库法）



注：扩容是成倍的。

1. 水平扩容表（双写迁移法）



第一步：（同步双写）修改应用配置和代码，加上双写，部署；

第二步：（同步双写）将老库中的老数据复制到新库中；

第三步：（同步双写）以老库为准校对新库中的老数据；第四步：（同步双写）修改应用配置和代码，去掉双写，部署；

注：双写是通用方案。

# 解决问题

## 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。

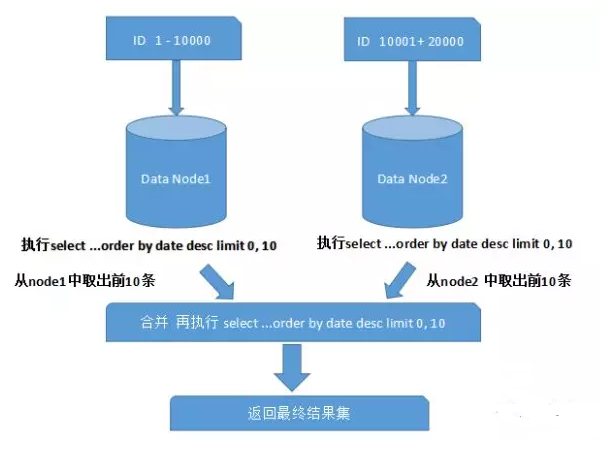
## 跨节点的count/order by/group by以及聚合函数问题

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。

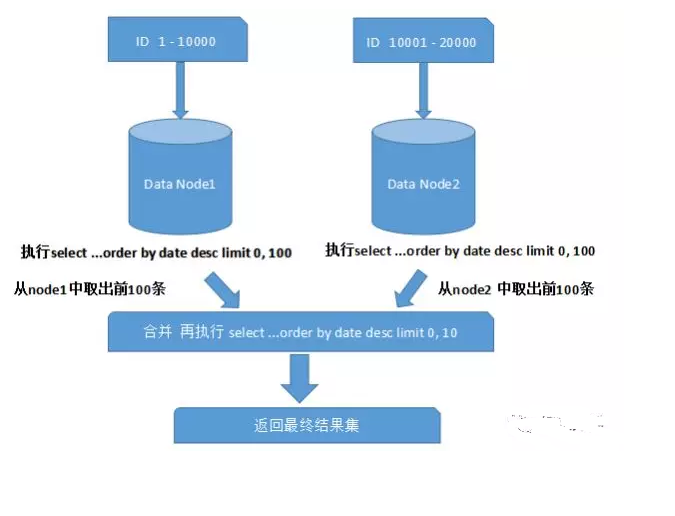
## 跨分片的排序分页

一般来讲，分页时需要按照指定字段进行排序。当排序字段就是分片字段的时候，我们通过分片规则可以比较容易定位到指定的分片，而当排序字段非分片字段的时候，情况就会变得比较复杂了。为了最终结果的准确性，我们需要在不同的分片节点中将数据进行排序并返回，并将不同分片返回的结果集进行汇总和再次排序，最后再返回给用户。如下图所示：



上面图中所描述的只是最简单的一种情况（取第一页数据），看起来对性能的影响并不大。但是，如果想取出第10页数据，情况又将变得复杂很多，如下图所示：

有些读者可能并不太理解，为什么不能像获取第一页数据那样简单处理（排序取出前10条再合并、排序）。其实并不难理解，因为各分片节点中的数据可能是随机的，为了排序的准确性，必须把所有分片节点的前N页数据都排序好后做合并，最后再进行整体的排序。很显然，这样的操作是比较消耗资源的，用户越往后翻页，系统性能将会越差。



那如何解决分库情况下的分页问题呢？有以下几种办法：

如果是在前台应用提供分页，则限定用户只能看前面n页，这个限制在业务上也是合理的，一般看后面的分页意义不大（如果一定要看，可以要求用户缩小范围重新查询）。

如果是后台批处理任务要求分批获取数据，则可以加大page size，比如每次获取5000条记录，有效减少分页数（当然离线访问一般走备库，避免冲击主库）。

分库设计时，一般还有配套大数据平台汇总所有分库的记录，有些分页查询可以考虑走大数据平台。

## ID问题

一旦数据库被切分到多个物理结点上，我们将不能再依赖数据库自身的主键生成机制。一方面，某个分区数据库自生成的ID无法保证在全局上是唯一的；另一方面，应用程序在插入数据之前需要先获得ID,以便进行SQL路由.

一些常见的主键生成策略

### UUID

使用UUID作主键是最简单的方案，但是缺点也是非常明显的。由于UUID非常的长，除占用大量存储空间外，最主要的问题是在索引上，在建立索引和基于索引进行查询时都存在性能问题。

**结合数据库维护一个Sequence表**

此方案的思路也很简单，在数据库中建立一个Sequence表，表的结构类似于：

CREATE TABLE `SEQUENCE` (

`table\_name` varchar(18) NOT NULL,

`nextid` bigint(20) NOT NULL,

PRIMARY KEY (`table\_name`)

) ENGINE=InnoDB

每当需要为某个表的新纪录生成ID时就从Sequence表中取出对应表的nextid，并将nextid的值加1后更新到数据库中以备下次使用。此方案也较简单，但缺点同样明显：由于所有插入任何都需要访问该表，该表很容易成为系统性能瓶颈，同时它也存在单点问题，一旦该表数据库失效，整个应用程序将无法工作。有人提出使用Master-Slave进行主从同步，但这也只能解决单点问题，并不能解决读写比为1:1的访问压力问题。

### Snowflake

在分布式系统中，需要生成全局UID的场合还是比较多的，twitter的snowflake解决了这种需求，实现也还是很简单的，除去配置信息，核心代码就是毫秒级时间41位 机器ID 10位 毫秒内序列12位。

10---0000000000 0000000000 0000000000 0000000000 0 --- 00000 ---00000 ---000000000000

在上面的字符串中，第一位为未使用（实际上也可作为long的符号位），接下来的41位为毫秒级时间，然后5位datacenter标识位，5位机器ID（并不算标识符，实际是为线程标识），然后12位该毫秒内的当前毫秒内的计数，加起来刚好64位，为一个Long型。

这样的好处是:整体上按照时间自增排序，并且整个分布式系统内不会产生ID碰撞（由datacenter和机器ID作区分），并且效率较高，经测试，snowflake每秒能够产生26万ID左右，完全满足需要。

## 数据迁移，容量规划，扩容等问题

来自淘宝综合业务平台团队，它利用对2的倍数取余具有向前兼容的特性（如对4取余得1的数对2取余也是1）来分配数据，避免了行级别的数据迁移，但是依然需要进行表级别的迁移，同时对扩容规模和分表数量都有限制。总得来说，这些方案都不是十分的理想，多多少少都存在一些缺点，这也从一个侧面反映出了Sharding扩容的难度。

## 事务

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。

### 分布式事务

参考：

关于分布式事务、两阶段提交、一阶段提交、Best Efforts 1PC模式和事务补偿机制的研究

优点：

基于两阶段提交，最大限度地保证了跨数据库操作的“原子性”，是分布式系统下最严格的事务实现方式。

实现简单，工作量小。由于多数应用服务器以及一些独立的分布式事务协调器做了大量的封装工作，使得项目中引入分布式事务的难度和工作量基本上可以忽略不计。

缺点：

系统“水平”伸缩的死敌。基于两阶段提交的分布式事务在提交事务时需要在多个节点之间进行协调,最大限度地推后了提交事务的时间点，客观上延长了事务的执行时间，这会导致事务在访问共享资源时发生冲突和死锁的概率增高，随着数据库节点的增多，这种趋势会越来越严重，从而成为系统在数据库层面上水平伸缩的"枷锁"， 这是很多Sharding系统不采用分布式事务的主要原因。

基于Best Efforts 1PC模式的事务

参考spring-data-neo4j的实现。鉴于Best Efforts 1PC模式的性能优势，以及相对简单的实现方式，它被大多数的sharding框架和项目采用

### 事务补偿（幂等值）

对于那些对性能要求很高，但对一致性要求并不高的系统，往往并不苛求系统的实时一致性，只要在一个允许的时间周期内达到最终一致性即可，这使得事务补偿机制成为一种可行的方案。

事务补偿机制最初被提出是在“长事务”的处理中，但是对于分布式系统确保一致性也有很好的参考意义。笼统地讲，与事务在执行中发生错误后立即回滚的方式不同，事务补偿是一种事后检查并补救的措施，它只期望在一个容许时间周期内得到最终一致的结果就可以了。

事务补偿的实现与系统业务紧密相关，并没有一种标准的处理方式。一些常见的实现方式有：对数据进行对帐检查;基于日志进行比对;定期同标准数据来源进行同步，等等。

## 分库策略

分库维度确定后，如何把记录分到各个库里呢?



**两种方式：**

根据数值范围，比如用户Id为1-9999的记录分到第一个库，10000-20000的分到第二个库，以此类推。

根据数值取模，比如用户Id mod n，余数为0的记录放到第一个库，余数为1的放到第二个库，以此类推。

**优劣比较：**

评价指标按照范围分库按照Mod分库

库数量前期数目比较小，可以随用户/业务按需增长前期即根据mode因子确定库数量，数目一般比较大

访问性能前期库数量小，全库查询消耗资源少，单库查询性能略差前期库数量大，全库查询消耗资源多，单库查询性能略好

调整库数量比较容易，一般只需为新用户增加库，老库拆分也只影响单个库困难，改变mod因子导致数据在所有库之间迁移

数据热点新旧用户购物频率有差异，有数据热点问题新旧用户均匀到分布到各个库，无热点

实践中，为了处理简单，选择mod分库的比较多。同时二次分库时，为了数据迁移方便，一般是按倍数增加，比如初始4个库，二次分裂为8个，再16个。这样对于某个库的数据，一半数据移到新库，剩余不动，对比每次只增加一个库，所有数据都要大规模变动。

补充下，mod分库一般每个库记录数比较均匀，但也有些数据库，存在超级Id，这些Id的记录远远超过其他Id，比如在广告场景下，某个大广告主的广告数可能占总体很大比例。如果按照广告主Id取模分库，某些库的记录数会特别多，对于这些超级Id，需要提供单独库来存储记录。

## 分库数量

分库数量首先和单库能处理的记录数有关，一般来说，Mysql 单库超过5000万条记录，Oracle单库超过1亿条记录，DB压力就很大(当然处理能力和字段数量/访问模式/记录长度有进一步关系)。

在满足上述前提下，如果分库数量少，达不到分散存储和减轻DB性能压力的目的；如果分库的数量多，好处是每个库记录少，单库访问性能好，但对于跨多个库的访问，应用程序需要访问多个库，如果是并发模式，要消耗宝贵的线程资源；如果是串行模式，执行时间会急剧增加。

最后分库数量还直接影响硬件的投入，一般每个分库跑在单独物理机上，多一个库意味多一台设备。所以具体分多少个库，要综合评估，一般初次分库建议分4-8个库。

## 路由透明

分库从某种意义上来说，意味着DB schema改变了，必然影响应用，但这种改变和业务无关，所以要尽量保证分库对应用代码透明，分库逻辑尽量在数据访问层处理。当然完全做到这一点很困难，具体哪些应该由DAL负责，哪些由应用负责，这里有一些建议：

对于单库访问，比如查询条件指定用户Id，则该SQL只需访问特定库。此时应该由DAL层自动路由到特定库，当库二次分裂时，也只要修改mod 因子，应用代码不受影响。

对于简单的多库查询，DAL负责汇总各个数据库返回的记录，此时仍对上层应用透明。

## 选择

使用框架还是自主研发？

目前市面上的分库分表中间件相对较多，其中基于代理方式的有MySQL Proxy和Amoeba，基于Hibernate框架的是Hibernate Shards，基于jdbc的有当当sharding-jdbc，基于mybatis的类似maven插件式的有蘑菇街的蘑菇街TSharding，通过重写spring的ibatis template类是Cobar Client，这些框架各有各的优势与短板，架构师可以在深入调研之后结合项目的实际情况进行选择，但是总的来说，我个人对于框架的选择是持谨慎态度的。一方面多数框架缺乏成功案例的验证，其成熟性与稳定性值得怀疑。另一方面，一些从成功商业产品开源出框架（如阿里和淘宝的一些开源项目）是否适合你的项目是需要架构师深入调研分析的。当然，最终的选择一定是基于项目特点、团队状况、技术门槛和学习成本等综合因素考量确定的。

# 企业级中间件

常用分库分表的中间件：

当当sharding-jdbc

蘑菇街TSharding

强悍重量级的中间件：

sharding

TDDL Smart Client的方式（淘宝）

Atlas(Qihoo 360)

alibaba.cobar(是阿里巴巴（B2B）部门开发)

MyCAT（基于阿里开源的Cobar产品而研发）

Oceanus(58同城数据库中间件)

OneProxy(支付宝首席架构师楼方鑫开发)

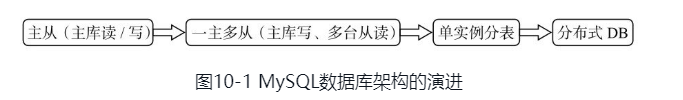
vitess（谷歌开发的数据库中间件）

## MyCAT

## OneProxy

### 背景

随着网站的壮大，MySQL数据库架构一般会经历如图10-1所示的演进过程。



当数据很小时，只用一台机器也许就能扛住访问压力；当数据量变大时，最初可以通过增加硬件（比如，加内存、换SSD硬盘，或者采购性能很强劲的小型机）的方法去解决。如果数据量越来越大，则最好从架构层出发进行改进，可以采用读/写分离的方式，同时采用多台slave备机提供读取业务，这样就降低了数据库的负载。

随着业务的进一步发展，一台主库的数据写入将成为瓶颈，如电商秒杀场景。依靠表的user\_id取模，把数据平均分散到不同的小表，再分布到各台机器上的方式，可以看成是数据迁移。

为什么要分库分表？原因如下：

❑ 单个库的数据容量太大，单个DB存储空间不够。

❑ 单个库表太多，查询时，打开表操作也消耗系统资源。

❑ 单个表容量太大，查询时，扫描行数过多，磁盘I/O大，查询缓慢。

❑ 单个库能承载的访问量有限，可高的访问量只能通过分库分表实现。

当一个表太大不利于维护时，可考虑将大表拆分成小表，当然，这些表是属于同一个数据库的，这种技术称为分表；当一个数据库的处理能力不够支撑业务，增加CPU的作用也十分有限时，就可能需要将部分表移到别的数据库，以增加系统处理能力，这种技术称为分库；通过精心的数据模型设计，将大的业务表拆分成小表，再将一系列小表分到不同的服务器，使得每台服务器都能独立处理部分业务，这种技术称为水平拆分，俗称分库分表。分表的数量可以和物理的机器数不一致，分表数量称为逻辑份数，分库的数量称为物理份数，当逻辑份数大于物理份数时，就可以迅速获得水平扩展能力。

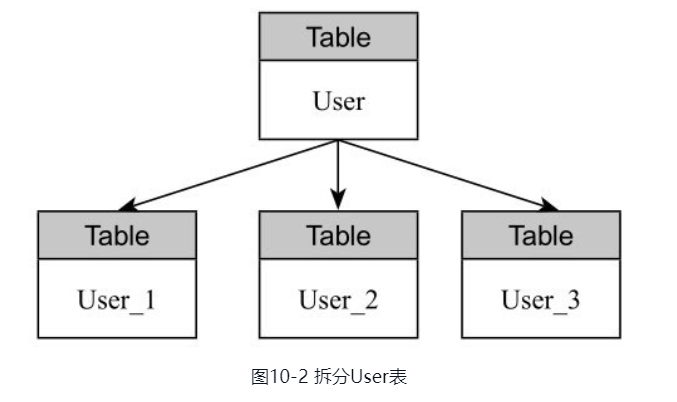
当使用传统商业数据库时，必须通过应用层修改代码来实现，现在流行的做法是将应用开发语言统一（比如用统一的Java框架）起来，然后编写统一的数据访问层（比如TDDL、ZDAL等）。这种做法在大并发量下的性能上有一定的优势，可以减少一次网络交互，但在开发上绑定了特定的开发语言，需要有强大的配置推送体系，并且需要有强大的运维团队来支持。当后台使用的是MySQL数据库，或兼容MySQL协议的数据库时，就可以不用修改应用程序，使用OneProxy来实现TDDL、ZDAL的功能，将后端的多台MySQL虚拟成一台MySQL提供给上层应用，对应用相对透明地实现分库分表的需求，从而快速获得MySQL上的水平扩展能力。

### 概述

OneProxy可以轻松实现MySQL的横向扩展，突破单台MySQL的限制，将数据库的同库分区表（范围分区、列表分区、哈希分区）扩展到跨库的分区表（将不同的分区放到不同的MySQL主从集群上），通过对单库事务及绝大部分跨库查询的透明支持，实现对应用相对非常透明的分布式数据库的架构支持，使得应用在进行少量修改的情况下就可以切换到分库分表的分布式互联网架构上，OneProxy内置的故障检测机制，可以让应用的开发变得非常简单。

对程序来讲，就是访问一张User表。后面怎么划分，按照什么规则划分，都不用管。例如，客户端插入user\_id:1-10十条数据，会通过OneProxy内置的SQL解释器分析SQL，按照range规则或hash规则，将客户端的请求自动分发到后端DB上，智能实现透明分库分表，每台DB上单独保存着分片后的数据。

分表后的示意图如图10-2所示。



### 限制

### 高可用

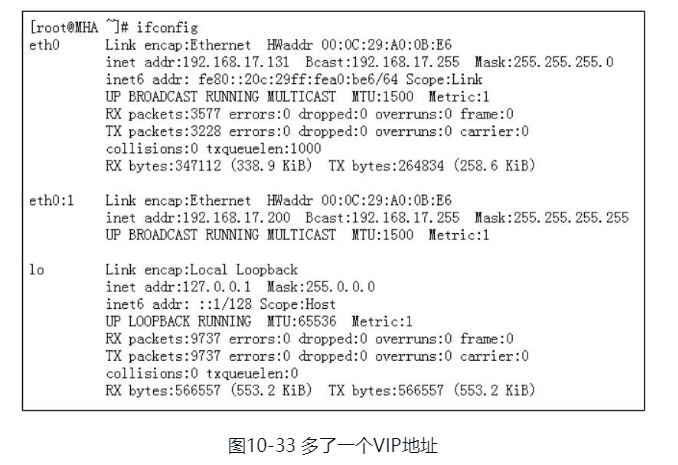
由于OneProxy内置了HA功能，因此可以不需要安装第三方软件实现高可用故障切换，也可以使用KeepAlived实现。如果将OneProxy部署在多台机器上，构成一个集群，那么应用就可以在程序中实现错误尝试机制，或者使用F5、HAProxy、LVS等软硬件做端口转发，这样，就可以根据一定的策略转发到任何一个OneProxy节点，从而做到OneProxy无单点服务。

1. 通过OneProxy内置功能实现高可用HA

假设两台OneProxy机器上的网卡名称为“eth0”，那么只需要在两台机器启动OneProxy的命令中新增一个参数“--vip-address=VIP地址/eth0:1”就可以了，OneProxy会自动检测VIP地址。如果一台机器重启，那么另一台机器会在1～2秒内自动接管VIP地址，以确保系统高可用，完全省去了多个复杂的集群软件的安全和配置，启动脚本如下：



启动脚本以后，通过ifconfig命令，可以查看到多了一个192.168.17.200 VIP地址，如图10-33所示。



1. 故障转移测试当kill掉oneproxy进程，或者把机器重启后，VIP会自动飘移到OneProxy备机上，切换时间为1～2秒。这里需要注意的是：故障切换是不打印日志的，如果想通过日志查看是何时切换的，目前的OneProxy 5.8.0版本（libevent: 2.0.22-stable）还不支持。

### 防火墙

# 互联网公司方案

## 蚂蚁金服

<https://mp.weixin.qq.com/s?__biz=MzU0ODg0OTIyNw==&mid=2247485843&idx=1&sn=12123eb11a4ac341c7e787422234bdab&source=41#wechat_redirect>

## TDSQL

<https://mp.weixin.qq.com/s?__biz=MzIwMzY1MzY4Mw==&mid=2247484577&idx=1&sn=c7801c597a4e6d524960167ad5505d75&chksm=96cd53fda1badaebbe5a9acdf5824c5039e04bd349f2c990f7dd8a8f5c294c5555f4ceadd7f3&scene=0&xtrack=1#rd>

## TiDB

# 应用场景

**对实时性要求比较高的场景，使用数据库的分区分表分库**。

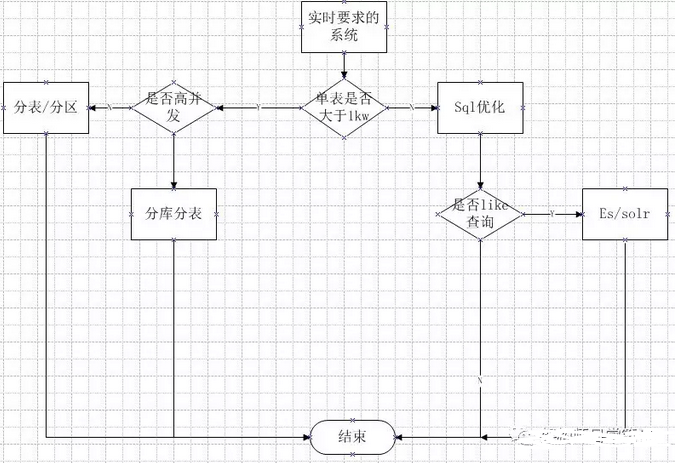
对实时性要求不高的场景，可以考虑使用索引库（ElasticSearch /solr）或者大数据hadoop平台来解决（如数据分析，挖掘，报表等）或者混合使用（如ElasticSearch +hbase/mongodb）。

分区解决冷热数据分离的问题;

分库解决互联网的**高并发**问题；

分表解决还联网的**高容量**问题；

分库分表解决高并发和高容量的问题。



**实时数据**

1、1000w以下的表通常不需要分库分表，如果数据增量比较小，可以分成冷热数据，考虑分区。

2、5000w以上的表通常要考虑进行分表(不考虑并发的情况)，此时以查询为主。

3、对并发要求比较高的系统，要考虑分库，此时分表解决不了并发的要求。

4、对大并发和高容量的可以使用分库分表方案解决

**非实时数据**

1、1000w以下的读可以通过从库来读取。

2、5000w以上的大量读可通过ElasticSearch /mongodb/hbase或者混合使用。

3、数据分析，挖掘，报表等最好不要和现有系统耦合，使用单独的库或者hadoop 平台来解决。