美团面试官：说说你对数据库分库分表的理解？

美团面试官：说说你对数据库分库分表的理解？

Java面试那些事儿

作者：butterfly100

来源：https://urlify.cn/JjuARf

 # 数据切分

关系型数据库本身比较容易成为系统瓶颈，单机存储容量、连接数、处理能力都有限。当单表的数据量达到1000W或100G以后，由于查询维度较多，即使添加从库、优化索引，做很多操作时性能仍下降严重。此时就要考虑对其进行切分了，切分的目的就在于减少数据库的负担，缩短查询时间。

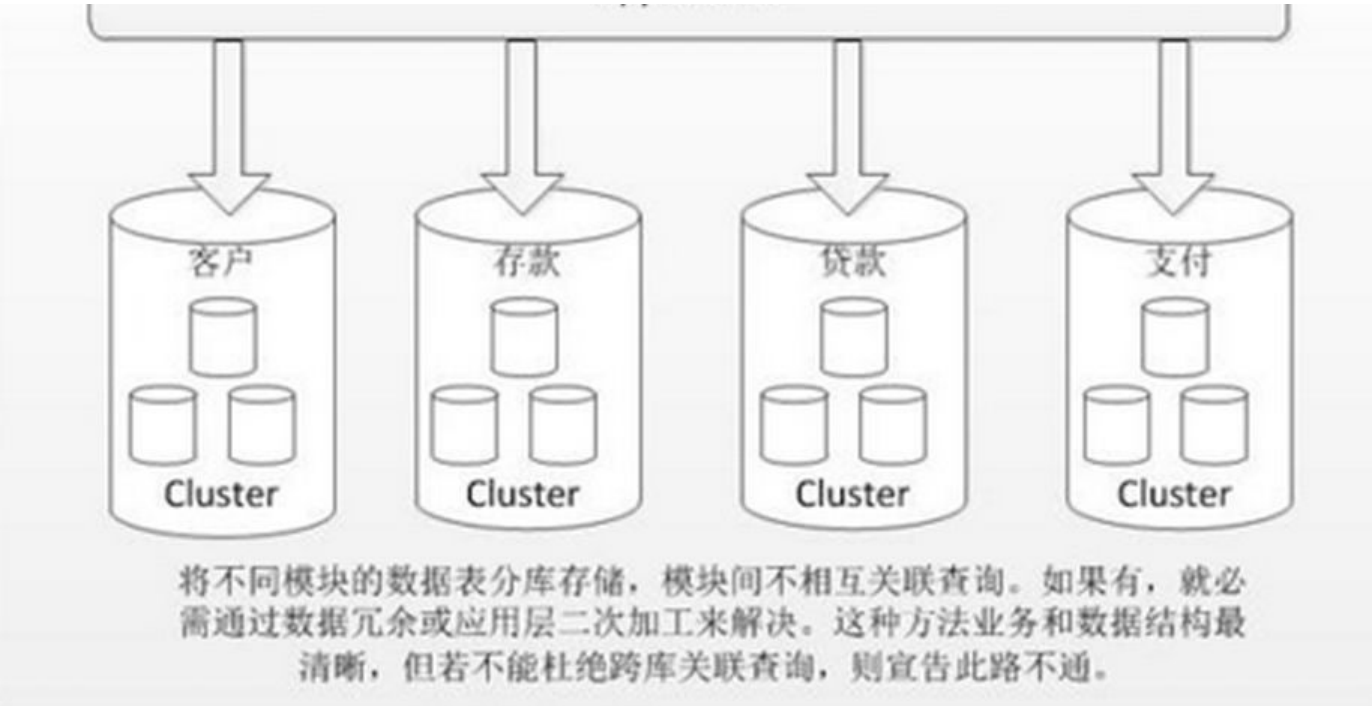
数据库分布式核心内容无非就是数据切分（Sharding），以及切分后对数据的定位、整合。数据切分就是将数据分散存储到多个数据库中，使得单一数据库中的数据量变小，通过扩充主机的数量缓解单一数据库的性能问题，从而达到提升数据库操作性能的目的。

数据切分根据其切分类型，可以分为两种方式：垂直（纵向）切分和水平（横向）切分

1、垂直（纵向）切分

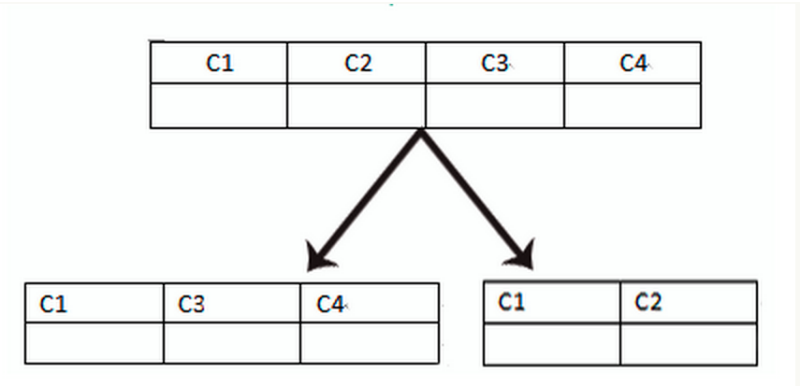
垂直切分常见有垂直分库和垂直分表两种。

垂直分库就是根据业务耦合性，将关联度低的不同表存储在不同的数据库。做法与大系统拆分为多个小系统类似，按业务分类进行独立划分。与"微服务治理"的做法相似，每个微服务使用单独的一个数据库。如图：



垂直分表是基于数据库中的"列"进行，某个表字段较多，可以新建一张扩展表，将不经常用或字段长度较大的字段拆分出去到扩展表中。在字段很多的情况下（例如一个大表有100多个字段），通过"大表拆小表"，更便于开发与维护，也能避免跨页问题，MySQL底层是通过数据页存储的，一条记录占用空间过大会导致跨页，造成额外的性能开销。

另外数据库以行为单位将数据加载到内存中，这样表中字段长度较短且访问频率较高，内存能加载更多的数据，命中率更高，减少了磁盘IO，从而提升了数据库性能。



垂直切分的优点：

- 解决业务系统层面的耦合，业务清晰
- 与微服务的治理类似，也能对不同业务的数据进行分级管理、维护、监控、扩展等
- 高并发场景下，垂直切分一定程度的提升IO、数据库连接数、单机硬件资源的瓶颈

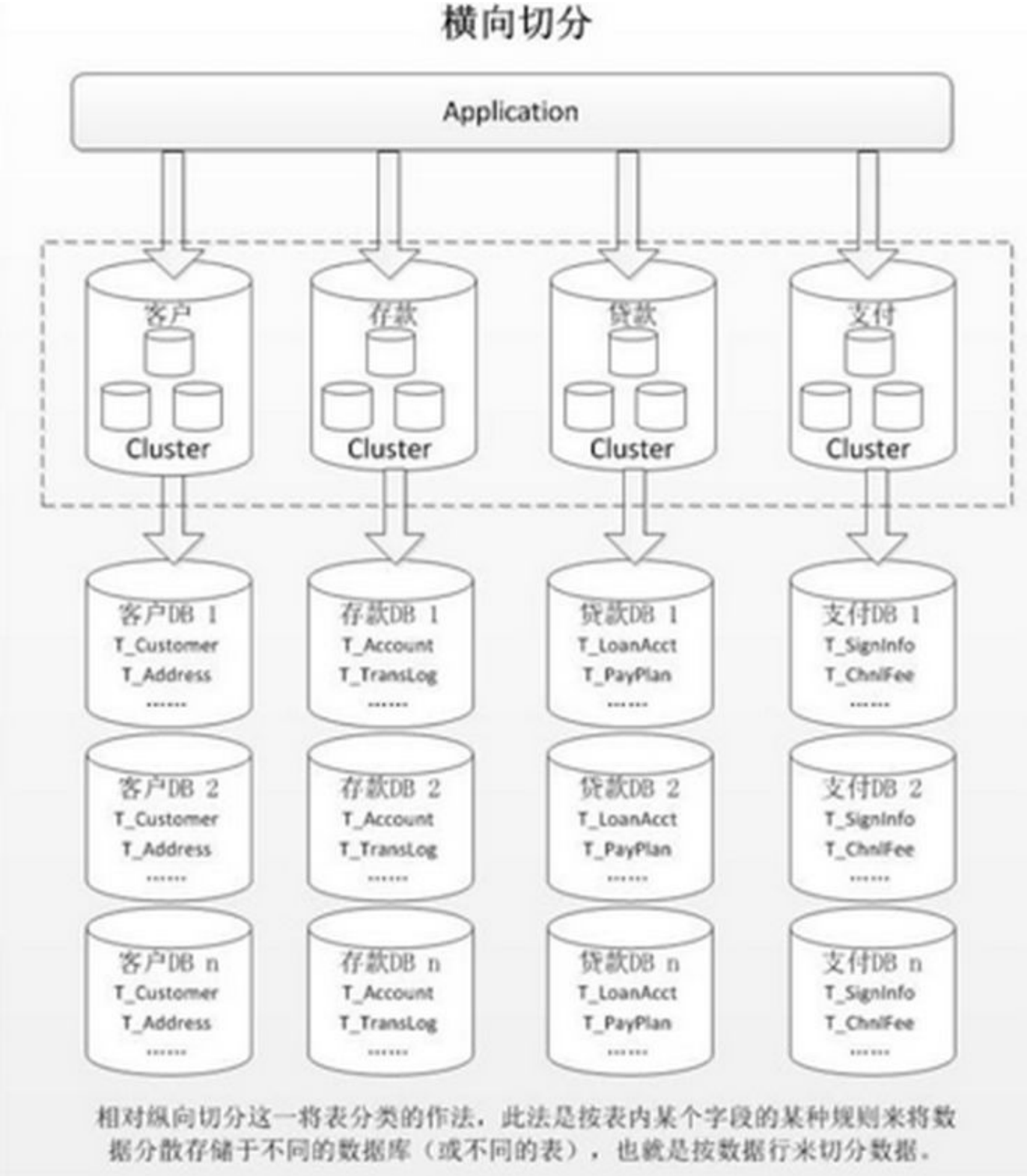
缺点：

- 部分表无法join，只能通过接口聚合方式解决，提升了开发的复杂度
- 分布式事务处理复杂
- 依然存在单表数据量过大的问题（需要水平切分）

2、水平（横向）切分

当一个应用难以再细粒度的垂直切分，或切分后数据量行数巨大，存在单库读写、存储性能瓶颈，这时候就需要进行水平切分了。

水平切分分为库内分表和分库分表，是根据表内数据内在的逻辑关系，将同一个表按不同的条件分散到多个数据库或多个表中，每个表中只包含一部分数据，从而使得单个表的数据量变小，达到分布式的效果。如图所示：



库内分表只解决了单一表数据量过大的问题，但没有将表分布到不同机器的库上，因此对于减轻MySQL数据库的压力来说，帮助不是很大，大家还是竞争同一个物理机的CPU、内存、网络IO，最好通过分库分表来解决。

水平切分的优点：

- 不存在单库数据量过大、高并发的性能瓶颈，提升系统稳定性和负载能力
- 应用端改造较小，不需要拆分业务模块

缺点：

- 跨分片的事务一致性难以保证
- 跨库的join关联查询性能较差
- 数据多次扩展难度和维护量极大

水平切分后同一张表会出现在多个数据库/表中，每个库/表的内容不同。几种典型的数据分片规则为：

1、根据数值范围

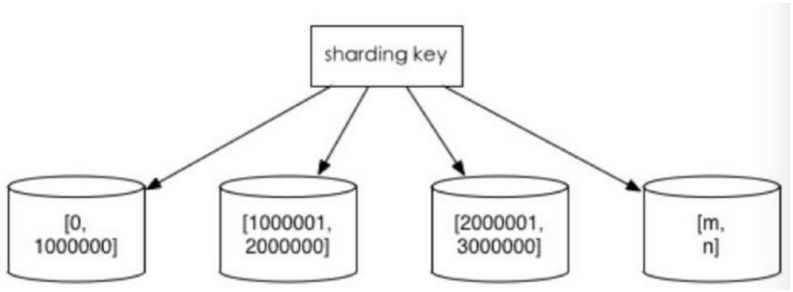
按照时间区间或ID区间来切分。例如：按日期将不同月甚至是日的数据分散到不同的库中；将userId为1~9999的记录分到第一个库，10000~20000的分到第二个库，以此类推。某种意义上，某些系统中使用的"冷热数据分离"，将一些使用较少的历史数据迁移到其他库中，业务功能上只提供热点数据的查询，也是类似的实践。

这样的优点在于：

- 单表大小可控
- 天然便于水平扩展，后期如果想对整个分片集群扩容时，只需要添加节点即可，无需对其他分片的数据进行迁移
- 使用分片字段进行范围查找时，连续分片可快速定位分片进行快速查询，有效避免跨分片查询的问题。

缺点：

- 热点数据成为性能瓶颈。连续分片可能存在数据热点，例如按时间字段分片，有些分片存储最近时间段内的数据，可能会被频繁的读写，而有些分片存储的历史数据，则很少被查询



2、根据数值取模

一般采用hash取模mod的切分方式，例如：将 Customer 表根据 cusno 字段切分到4个库中，余数为0的放到第一个库，余数为1的放到第二个库，以此类推。这样同一个用户的数据会

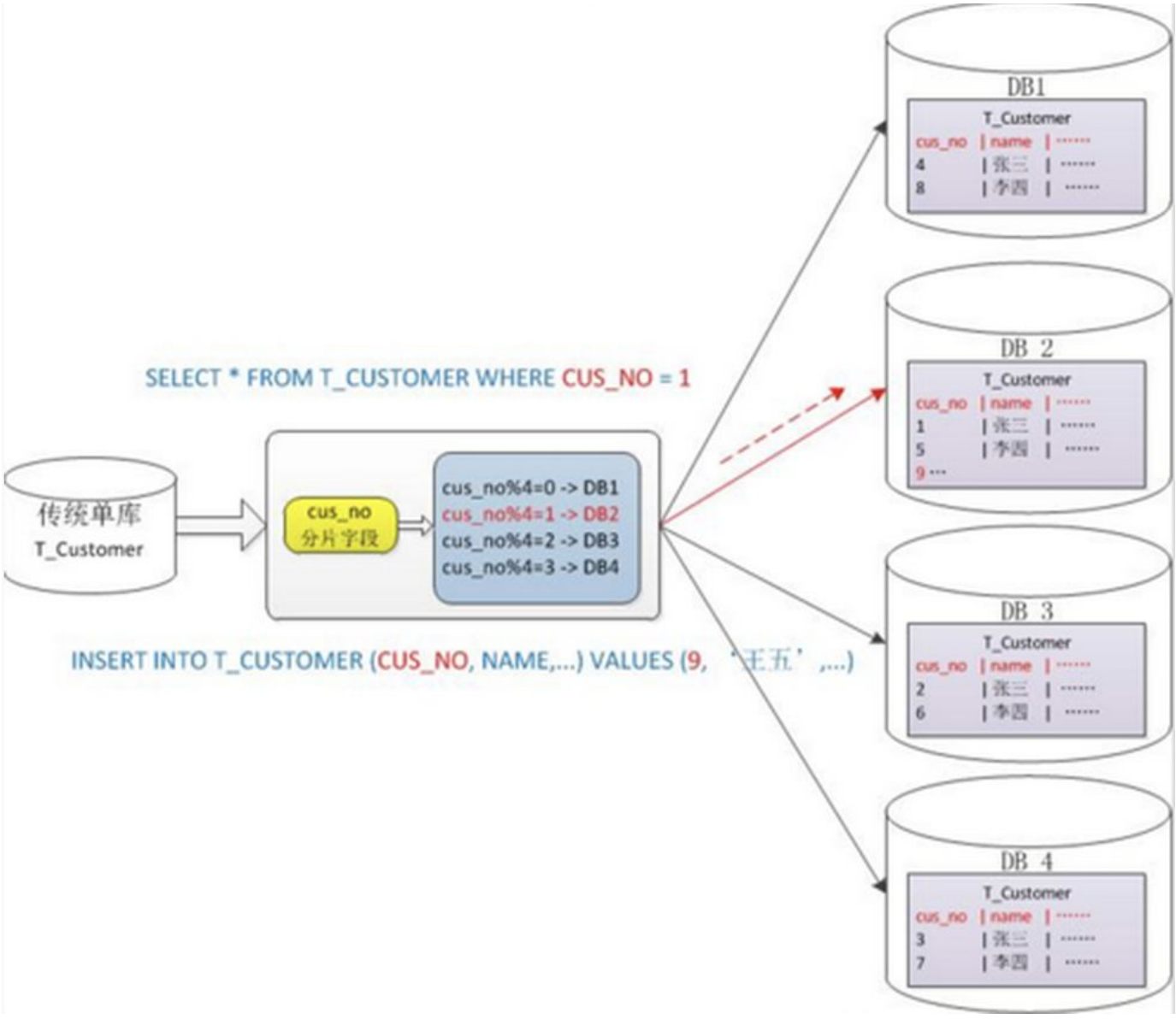
分散到同一个库中，如果查询条件带有cusno字段，则可明确定位到相应库去查询。

优点：

- 数据分片相对比较均匀，不容易出现热点和并发访问的瓶颈

缺点：

- 后期分片集群扩容时，需要迁移旧的数据（使用一致性hash算法能较好的避免这个问题）
- 容易面临跨分片查询的复杂问题。比如上例中，如果频繁用到的查询条件中不带cusno时，将会导致无法定位数据库，从而需要同时向4个库发起查询，再在内存中合并数据，取最小集返回给应用，分库反而成为拖累。



 # 分库分表带来的问题

分库分表能有效的环节单机和单库带来的性能瓶颈和压力，突破网络IO、硬件资源、连接数的瓶颈，同时也带来了一些问题。下面将描述这些技术挑战以及对应的解决思路。

1、事务一致性问题

分布式事务

当更新内容同时分布在不同库中，不可避免会带来跨库事务问题。跨分片事务也是分布式事务，没有简单的方案，一般可使用"XA协议"和"两阶段提交"处理。

分布式事务能最大限度保证了数据库操作的原子性。但在提交事务时需要协调多个节点，推后了提交事务的时间点，延长了事务的执行时间。导致事务在访问共享资源时发生冲突或死锁的概率增高。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扩展的枷锁。

最终一致性

对于那些性能要求很高，但对一致性要求不高的系统，往往不苛求系统的实时一致性，只要在允许的时间段内达到最终一致性即可，可采用事务补偿的方式。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补救的措施，一些常见的实现方法有：对数据进行对账检查，基于日志进行对比，定期同标准数据来源进行同步等等。事务补偿还要结合业务系统来考虑。

2、跨节点关联查询 join 问题

切分之前，系统中很多列表和详情页所需的数据可以通过sql join来完成。而切分之后，数据可能分布在不同的节点上，此时join带来的问题就比较麻烦了，考虑到性能，尽量避免使用join查询。

解决这个问题的一些方法：

1）全局表

全局表，也可看做是"数据字典表"，就是系统中所有模块都可能依赖的一些表，为了避免跨库join查询，可以将这类表在每个数据库中都保存一份。这些数据通常很少会进行修改，所以也不担心一致性的问题。

2）字段冗余

一种典型的反范式设计，利用空间换时间，为了性能而避免join查询。例如：订单表保存userId时候，也将userName冗余保存一份，这样查询订单详情时就不需要再去查询"买家user表"了。

但这种方法适用场景也有限，比较适用于依赖字段比较少的情况。而冗余字段的数据一致性也较难保证，就像上面订单表的例子，买家修改了userName后，是否需要在历史订单中同步更

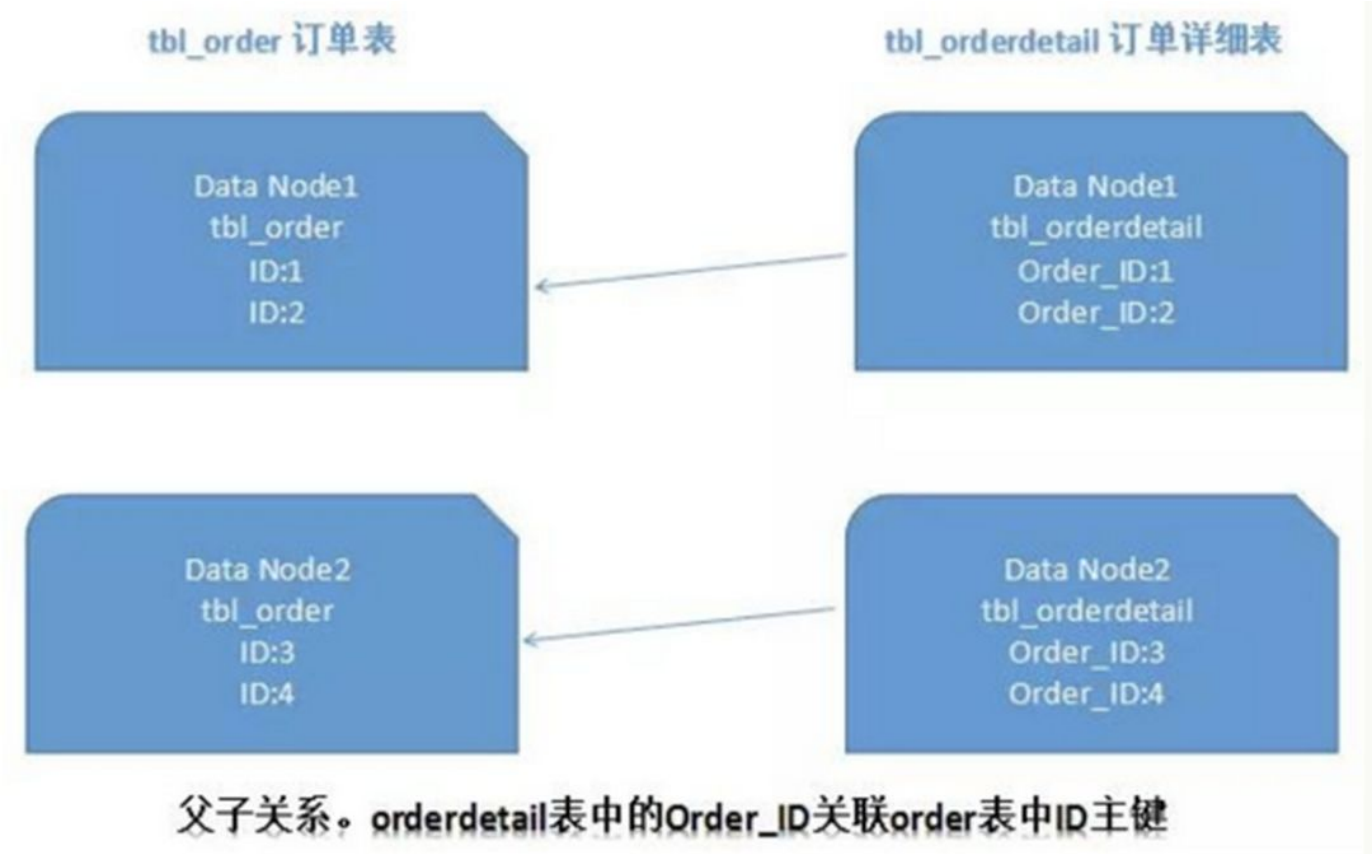
新呢？这也要结合实际业务场景进行考虑。

3）数据组装

在系统层面，分两次查询，第一次查询的结果集中找出关联数据id，然后根据id发起第二次请求得到关联数据。最后将获得到的数据进行字段拼装。

4）ER分片

关系型数据库中，如果可以先确定表之间的关联关系，并将那些存在关联关系的表记录存放在同一个分片上，那么就能较好的避免跨分片join问题。在1:1或1:n的情况下，通常按照主表的ID主键切分。如下图所示：



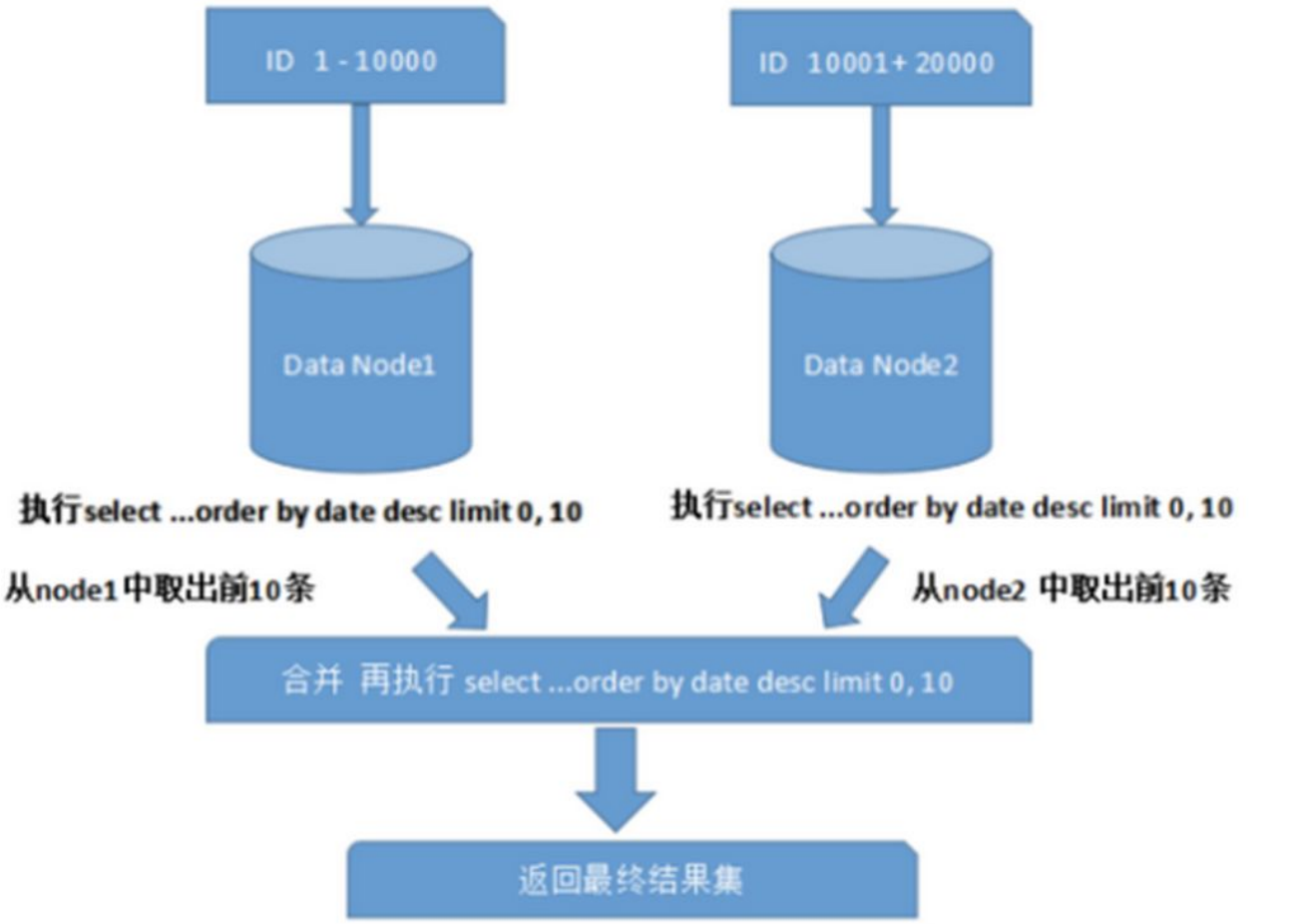
这样一来，Data Node1上面的order订单表与orderdetail订单详情表就可以通过orderId进行局部的关联查询了，Data Node2上也一样。

3、跨节点分页、排序、函数问题

跨节点多库进行查询时，会出现limit分页、order by排序等问题。分页需要按照指定字段进行排序，当排序字段就是分片字段时，通过分片规则就比较容易定位到指定的分片；当排序字段非分片字段时，就变得比较复杂了。

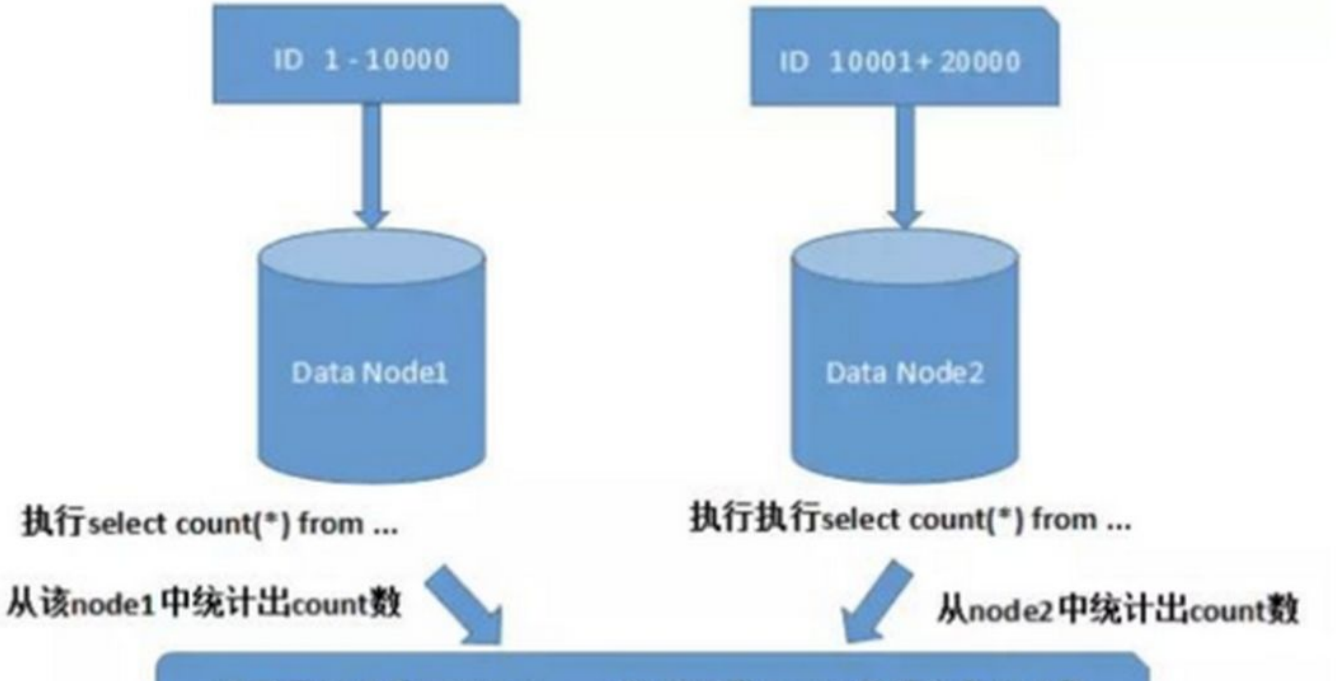
需要先在不同的分片节点中将数据进行排序并返回，然后将不同分片返回的结果集进行汇总和

再次排序，最终返回给用户。如图所示：



上图中只是取第一页的数据，对性能影响还不是很大。但是如果取得页数很大，情况则变得复杂很多，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，需要将所有节点的前N页数据都排序好做合并，最后再进行整体的排序，这样的操作时很耗费CPU和内存资源的，所以页数越大，系统的性能也会越差。

在使用Max、Min、Sum、Count之类的函数进行计算的时候，也需要先在每个分片上执行相应的函数，然后将各个分片的结果集进行汇总和再次计算，最终将结果返回。如图所示：



4、全局主键避重问题

在分库分表环境中，由于表中数据同时存在不同数据库中，主键值平时使用的自增长将无用武之地，某个分区数据库自生成的ID无法保证全局唯一。因此需要单独设计全局主键，以避免跨库主键重复问题。有一些常见的主键生成策略：

1）UUID

UUID标准形式包含32个16进制数字，分为5段，形式为8-4-4-4-12的36个字符，
例如：550e8400-e29b-41d4-a716-446655440000
UUID是主键是最简单的方案，本地生成，性能高，没有网络耗时。但缺点也很明显，由于UUID非常长，会占用大量的存储空间；另外，作为主键建立索引和基于索引进行查询时都会存在性能问题，在InnoDB下，UUID的无序性会引起数据位置频繁变动，导致分页。

2）结合数据库维护主键ID表

在数据库中建立 sequence 表：

```
CREATE TABLE `sequence` (
  `id` bigint(20) unsigned NOT NULL auto_increment,
  `stub` char(1) NOT NULL default '',
  PRIMARY KEY  (`id`),
  UNIQUE KEY `stub` (`stub`)
) ENGINE=MyISAM;
```

stub字段设置为唯一索引，同一stub值在sequence表中只有一条记录，可以同时为多张表生成全局ID。sequence表的内容，如下所示：

```
+-------------------+------+
| id                | stub |
+-------------------+------+
| 72157623227190423 |    a |
+-------------------+------+
```

使用 MyISAM 存储引擎而不是 InnoDB，以获取更高的性能。MyISAM使用的是表级别的锁，对表的读写是串行的，所以不用担心在并发时两次读取同一个ID值。

当需要全局唯一的64位ID时，执行：

```
1  REPLACE INTO sequence (stub) VALUES ('a');
2  SELECT LAST_INSERT_ID();
```

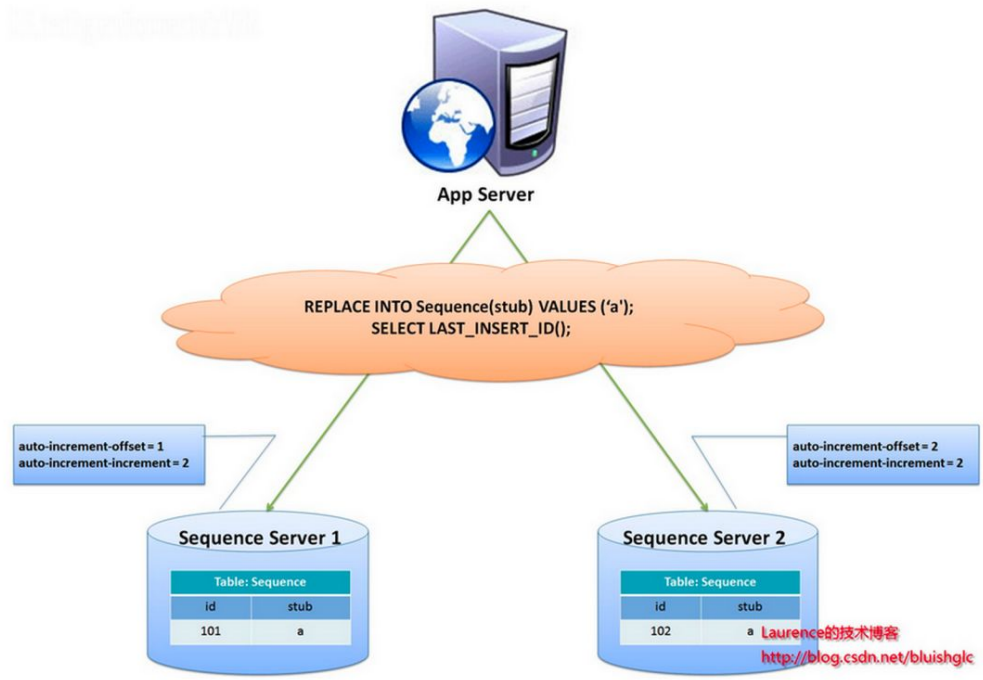
这两条语句是Connection级别的，select last_insert_id() 必须与 replace into 在同一数据库连接下才能得到刚刚插入的新ID。

使用replace into代替insert into好处是避免了表行数过大，不需要另外定期清理。

此方案较为简单，但缺点也明显：存在单点问题，强依赖DB，当DB异常时，整个系统都不可用。配置主从可以增加可用性，但当主库挂了，主从切换时，数据一致性在特殊情况下难以保证。另外性能瓶颈限制在单台MySQL的读写性能。

flickr团队使用的一种主键生成策略，与上面的sequence表方案类似，但更好的解决了单点和性能瓶颈的问题。

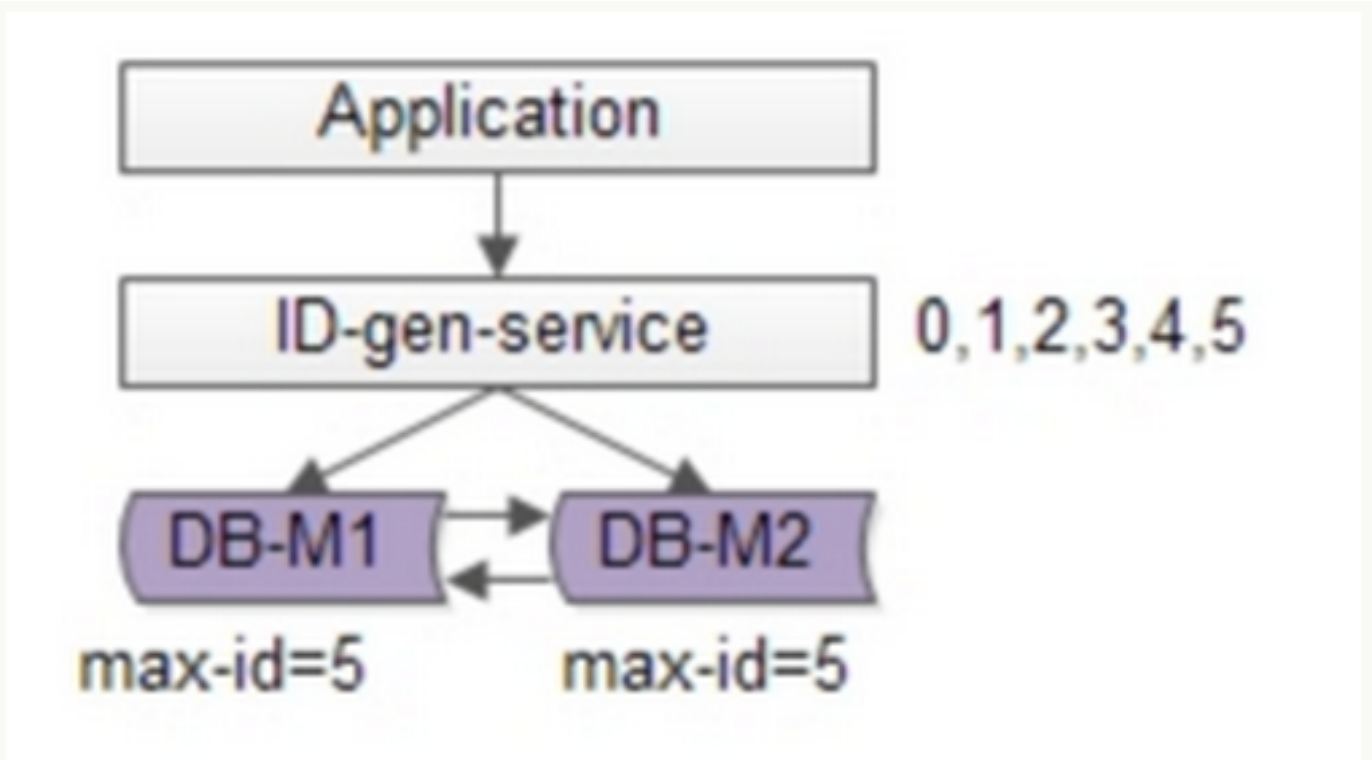
这一方案的整体思想是：建立2个以上的全局ID生成的服务器，每个服务器上只部署一个数据库，每个库有一张sequence表用于记录当前全局ID。表中ID增长的步长是库的数量，起始值依次错开，这样能将ID的生成散列到各个数据库上。如下图所示：



由两个数据库服务器生成ID，设置不同的auto_increment值。第一台sequence的起始值为1，每次步长增长2，另一台的sequence起始值为2，每次步长增长也是2。结果第一台生成的ID都是奇数（1, 3, 5, 7 ...），第二台生成的ID都是偶数（2, 4, 6, 8 ...）。

这种方案将生成ID的压力均匀分布在两台机器上。同时提供了系统容错，第一台出现了错误，可以自动切换到第二台机器上获取ID。但有以下几个缺点：系统添加机器，水平扩展时较复杂；每次获取ID都要读写一次DB，DB的压力还是很大，只能靠堆机器来提升性能。

可以基于flickr的方案继续优化，使用批量的方式降低数据库的写压力，每次获取一段区间的ID号段，用完之后再去数据库获取，可以大大减轻数据库的压力。如下图所示：

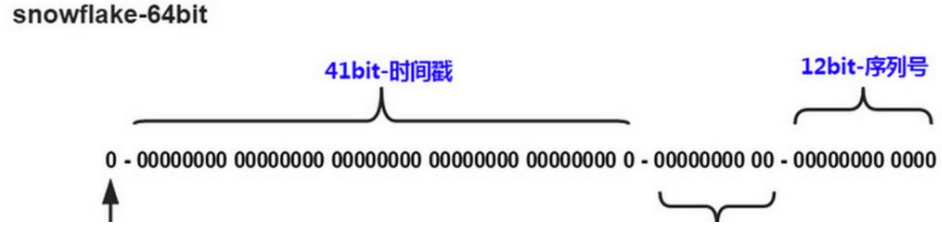


还是使用两台DB保证可用性，数据库中只存储当前的最大ID。ID生成服务每次批量拉取6个ID，先将max_id修改为5，当应用访问ID生成服务时，就不需要访问数据库，从号段缓存中依次派发0~5的ID。当这些ID发完后，再将max_id修改为11，下次就能派发6~11的ID。于是，数据库的压力降低为原来的1/6。

3）Snowflake分布式自增ID算法

Twitter的snowflake算法解决了分布式系统生成全局ID的需求，生成64位的Long型数字，组成部分：

- 第一位未使用
- 接下来41位是毫秒级时间，41位的长度可以表示69年的时间
- 5位datacenterId，5位workerId。10位的长度最多支持部署1024个节点
- 最后12位是毫秒内的计数，12位的计数顺序号支持每个节点每毫秒产生4096个ID序列



这样的好处是：毫秒数在高位，生成的ID整体上按时间趋势递增；不依赖第三方系统，稳定性和效率较高，理论上QPS约为409.6w/s（1000*2^12），并且整个分布式系统内不会产生ID碰撞；可根据自身业务灵活分配bit位。

不足就在于：强依赖机器时钟，如果时钟回拨，则可能导致生成ID重复。

综上

结合数据库和snowflake的唯一ID方案，可以参考业界较为成熟的解法：Leaf——美团点评分布式ID生成系统，并考虑到了高可用、容灾、分布式下时钟等问题。

5、数据迁移、扩容问题

当业务高速发展，面临性能和存储的瓶颈时，才会考虑分片设计，此时就不可避免的需要考虑历史数据迁移的问题。一般做法是先读出历史数据，然后按指定的分片规则再将数据写入到各个分片节点中。此外还需要根据当前的数据量和QPS，以及业务发展的速度，进行容量规划，推算出大概需要多少分片（一般建议单个分片上的单表数据量不超过1000W）

如果采用数值范围分片，只需要添加节点就可以进行扩容了，不需要对分片数据迁移。如果采用的是数值取模分片，则考虑后期的扩容问题就相对比较麻烦。

 # 什么时候考虑切分

下面讲述一下什么时候需要考虑做数据切分。

1、能不切分尽量不要切分

并不是所有表都需要进行切分，主要还是看数据的增长速度。切分后会在某种程度上提升业务的复杂度，数据库除了承载数据的存储和查询外，协助业务更好的实现需求也是其重要工作之一。

不到万不得已不用轻易使用分库分表这个大招，避免"过度设计"和"过早优化"。分库分表之前，不要为分而分，先尽力去做力所能及的事情，例如：升级硬件、升级网络、读写分离、索引优化等等。当数据量达到单表的瓶颈时候，再考虑分库分表。

2、数据量过大，正常运维影响业务访问

这里说的运维，指：

1）对数据库备份，如果单表太大，备份时需要大量的磁盘IO和网络IO。例如1T的数据，网络传输占50MB时候，需要20000秒才能传输完毕，整个过程的风险都是比较高的

2）对一个很大的表进行DDL修改时，MySQL会锁住全表，这个时间会很长，这段时间业务不能访问此表，影响很大。如果使用pt-online-schema-change，使用过程中会创建触发器和影子表，也需要很长的时间。在此操作过程中，都算为风险时间。将数据表拆分，总量减少，有助于降低这个风险。

3）大表会经常访问与更新，就更有可能出现锁等待。将数据切分，用空间换时间，变相降低访问压力

3、随着业务发展，需要对某些字段垂直拆分

举个例子，假如项目一开始设计的用户表如下：

```
id                  bigint          #用户的ID
name                varchar         #用户的名字
last_login_time     datetime        #最近登录时间
personal_info       text            #私人信息
.....                               #其他信息字段
```

在项目初始阶段，这种设计是满足简单的业务需求的，也方便快速迭代开发。而当业务快速发展时，用户量从10w激增到10亿，用户非常的活跃，每次登录会更新 last_login_name 字段，使得 user 表被不断update，压力很大。而其他字段：id, name, personal_info 是不变的或很少更新的，此时在业务角度，就要将 last_login_time 拆分出去，新建一个 user_time 表。

personal_info 属性是更新和查询频率较低的，并且text字段占据了太多的空间。这时候，就要对此垂直拆分出 user_ext 表了。

4、数据量快速增长

随着业务的快速发展，单表中的数据量会持续增长，当性能接近瓶颈时，就需要考虑水平切分，做分库分表了。此时一定要选择合适的切分规则，提前预估好数据容量

5、安全性和可用性

鸡蛋不要放在一个篮子里。在业务层面上垂直切分，将不相关的业务的数据库分隔，因为每个

业务的数据量、访问量都不同，不能因为一个业务把数据库搞挂而牵连到其他业务。利用水平切分，当一个数据库出现问题时，不会影响到100%的用户，每个库只承担业务的一部分数据，这样整体的可用性就能提高。

 # 案例分析

1、用户中心业务场景

用户中心是一个非常常见的业务，主要提供用户注册、登录、查询/修改等功能，其核心表为：

```
User(uid, login_name, passwd, sex, age, nickname)

uid为用户ID,  主键
login_name, passwd, sex, age, nickname,  用户属性
```

任何脱离业务的架构设计都是耍流氓，在进行分库分表前，需要对业务场景需求进行梳理：
1.用户侧：前台访问，访问量较大，需要保证高可用和高一致性。主要有两类需求：
用户登录：通过login_name/phone/email查询用户信息，1%请求属于这种类型用户信息查询：登录之后，通过uid来查询用户信息，99%请求属这种类型

2.运营侧：后台访问，支持运营需求，按照年龄、性别、登陆时间、注册时间等进行分页的查询。是内部系统，访问量较低，对可用性、一致性的要求不高。

2、水平切分方法

当数据量越来越大时，需要对数据库进行水平切分，上文描述的切分方法有"根据数值范围"和"根据数值取模"。

"根据数值范围"：以主键uid为划分依据，按uid的范围将数据水平切分到多个数据库上。例如：user-db1存储uid范围为0~1000w的数据，user-db2存储uid范围为1000w~2000wuid数据。

优点是：扩容简单，如果容量不够，只要增加新db即可。
不足是：请求量不均匀，一般新注册的用户活跃度会比较高，所以新的user-db2会比user-db1负载高，导致服务器利用率不平衡

"根据数值取模"：也是以主键uid为划分依据，按uid取模的值将数据水平切分到多个数据库上。例如：user-db1存储uid取模得1的数据，user-db2存储uid取模得0的uid数据。

优点是：数据量和请求量分布均均匀
不足是：扩容麻烦，当容量不够时，新增加db，需要rehash。需要考虑对数据进行平滑的迁移。

3、非uid的查询方法

水平切分后，对于按uid查询的需求能很好的满足，可以直接路由到具体数据库。而按非uid的查询，例如login_name，就不知道具体该访问哪个库了，此时需要遍历所有库，性能会降低很多。

对于用户侧，可以采用"建立非uid属性到uid的映射关系"的方案；对于运营侧，可以采用"前台与后台分离"的方案。

建立非uid属性到uid的映射关系

1）映射关系

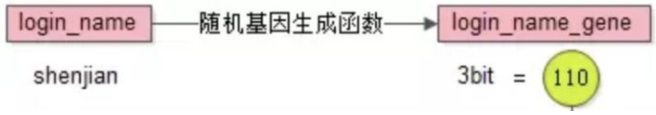
例如：login_name不能直接定位到数据库，可以建立login_name→uid的映射关系，用索引表或缓存来存储。当访问login_name时，先通过映射表查询出login_name对应的uid，再通过uid定位到具体的库。

映射表只有两列，可以承载很多数据，当数据量过大时，也可以对映射表再做水平切分。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。

2）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。



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。

 # 支持分库分表中间件

站在巨人的肩膀上能省力很多，目前分库分表已经有一些较为成熟的开源解决方案：

- sharding-jdbc（当当）
- TSharding（蘑菇街）
- Atlas（奇虎360）
- Cobar（阿里巴巴）
- MyCAT（基于Cobar）
- Oceanus（58同城）
- Vitess（谷歌）

Java面试那些事儿

微信扫描二维码，关注我的公众号

Java面试那些事儿

觉得不错，请给个「在看」
分享给你的朋友！

👎**点我**，查看更多精彩文章。

阅读原文