Mysql在大型网站的应用架构演变

张晓龙

来自：运维部 time 2014-06-16 16:36 次 1

本文链接地址：Mysql在大型网站的应用架构演变

写在最前：

本文主要描述在网站的不同的并发访问量级下，Mysql架构的演变

可扩展性

架构的可扩展性往往和并发是息息相关，没有并发的增长，也就没有必要做高可扩展性的架构，这里对可扩展性进行简单介绍一下，常用的扩展手段有以下两种

Scale-up： 纵向扩展，通过替换为更好的机器和资源来实现伸缩，提升服务能力

Scale-out：横向扩展，通过加节点（机器）来实现伸缩，提升服务能力

对于互联网的高并发应用来说，无疑**Scale out才是出路**，通过纵向的买更高端的机器一直是我们所避讳的问题，也不是长久之计，在scale out的理论下，可扩展性的理想状态是什么?

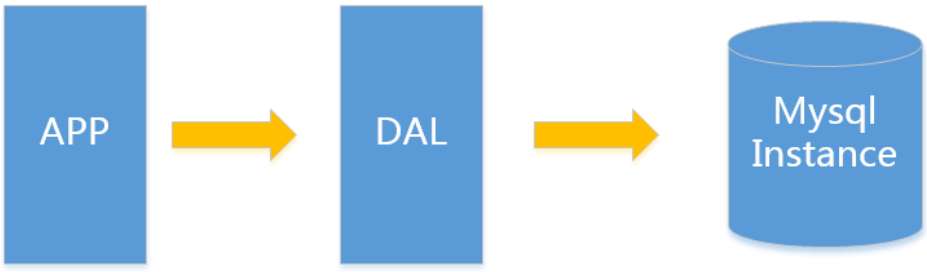
可扩展性的理想状态

一个服务，当面临更高的并发的时候，**能够通过简单增加机器来提升服务支撑的并发度，且增加机器过程中对线上服务无影响(no down time)，这就是可扩展性的理想状态!**

架构的演变

V1.0 简单网站架构

一个简单的小型网站或者应用背后的架构可以非常简单，数据存储只需要一个mysql instance就能满足数据读取和写入需求（这里忽略掉了数据备份的实例），处于这个时间段的网站，一般会把所有的信息存到一个database instance里面。



在这样的架构下，我们来看看数据存储的瓶颈是什么?

1.数据量的总大小 一个机器放不下时

2.数据的索引（B+ Tree）一个机器的内存放不下时

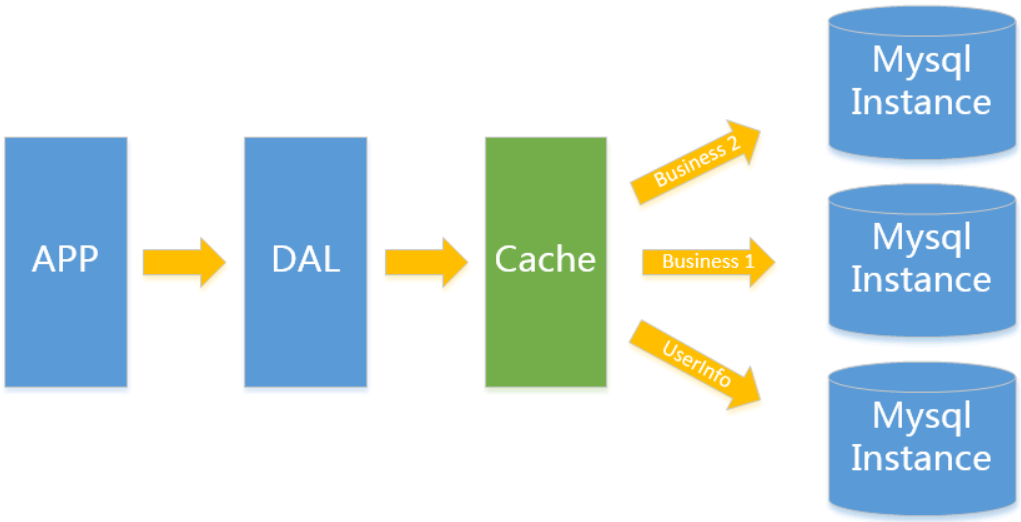
3.访问量(读写混合)一个实例不能承受

只有当以上3件事情任何一件或多件满足时，我们才需要考虑往下一级演变。 从此我们可以看出，事实上对于很多小公司小应用，这种架构已经足够满足他们的需求了，初期数据量的准确评估是杜绝过度设计很重要的一环，毕竟没有人愿意为不可能发生的事情而浪费自己的经历。

这里简单举个我的例子，对于用户信息这类表 （3个索引），16G内存能放下大概2000W行数据的索引，简单的读和写混合访问量3000/s左右没有问题，你的应用场景是否

V2.0 垂直拆分

一般当V1.0 遇到瓶颈时，首先最简便的拆分方法就是垂直拆分，何谓垂直？就是从业务角度来看，将关联性不强的数据拆分到不同的instance上，从而达到消除瓶颈的目标。以图中的为例，将用户信息数据，和业务数据拆分到不同的三个实例上。对于重复读类型比较多的场景，我们还可以加一层cache，来减少对DB的压力。



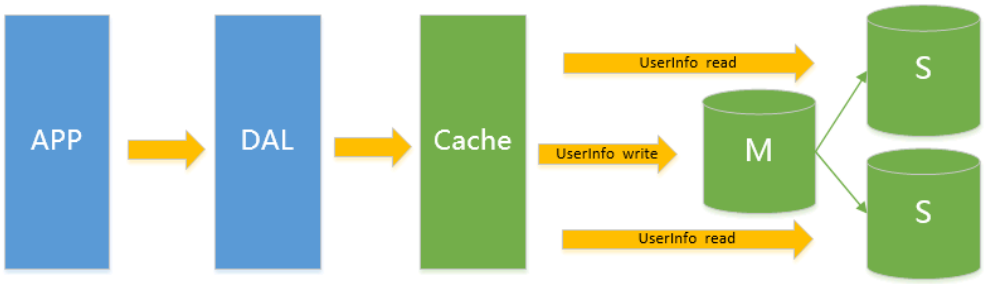
在这样的架构下，我们来看看数据存储的瓶颈是什么？

1.单实例单业务 依然存在V1.0所述瓶颈

遇到瓶颈时可以考虑往本文更高V版本升级，若是读请求导致达到性能瓶颈可以考虑往V3.0升级， 其他瓶颈考虑往V4.0升级

V3.0 主从架构

此类架构主要解决V2.0架构下的读问题，通过给Instance挂数据实时备份的思路来迁移读取的压力，在Mysql的场景下就是通过主从结构，主库抗写压力，通过从库来分担读压力，对于写少读多的应用，V3.0主从架构完全能够胜任

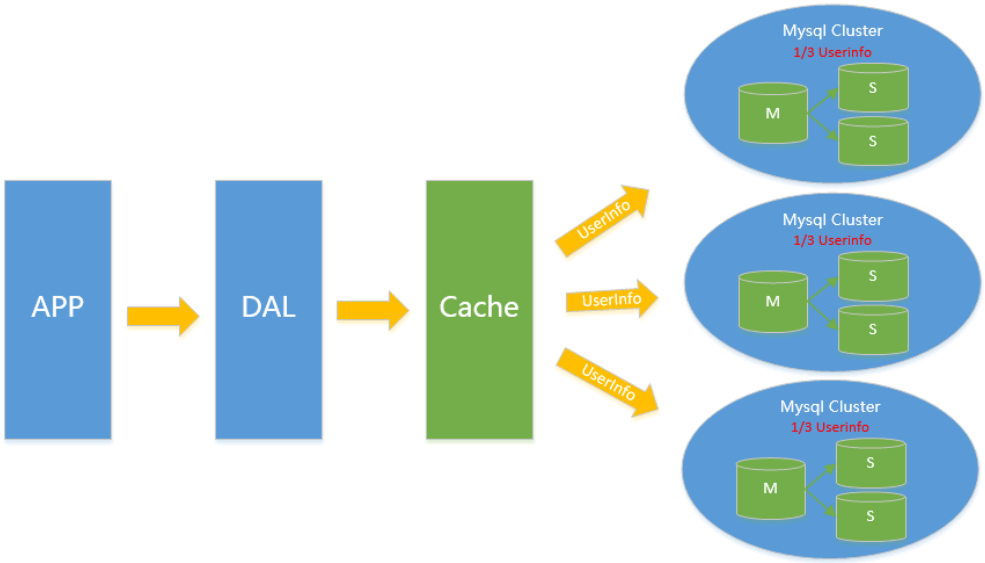


在这样的架构下，我们来看看数据存储的瓶颈是什么？

1.写入量主库不能承受

V4.0 水平拆分

对于V2.0 V3.0方案遇到瓶颈时，都可以通过水平拆分来解决，水平拆分和垂直拆分有较大区别，垂直拆分拆完的结果，在一个实例上是拥有全量数据的，而水平拆分之后，任何实例都只有全量的1/n的数据，以下图Userinfo的拆分为例，将userinfo拆分为3个cluster，每个cluster持有总量的1/3数据，3个cluster数据的总和等于一份完整数据（注：这里不再叫单个实例 而是叫一个cluster 代表包含主从的一个小mysql集群）



数据如何路由?

1.Range拆分

sharding key按连续区间段路由，一般用在有严格自增ID需求的场景上，如Userid, Userid Range的小例子：以userid 3000W 为Range进行拆分　1号cluster　userid 1-3000W　2号cluster　userid　3001W-6000W

2.List拆分

List拆分与Range拆分思路一样，都是通过给不同的sharding key来路由到不同的cluster,但是具体方法有些不同,List主要用来做sharding key不是连续区间的序列落到一个cluster的情况，如以下场景：

假定有20个音像店，分布在4个有经销权的地区，如下表所示：

地区

商店ID 号

北区

3, 5, 6, 9, 17

东区

1, 2, 10, 11, 19, 20

西区

4, 12, 13, 14, 18

中心区

7, 8, 15, 16

业务希望能够把一个地区的所有数据组织到一起来搜索，这种场景List拆分可以轻松搞定

3.Hash拆分

通过对sharding key 进行哈希的方式来进行拆分，常用的哈希方法有除余,字符串哈希等等，除余如按userid%n 的值来决定数据读写哪个cluster，其他哈希类算法这里就不细展开讲了。

数据拆分后引入的问题：

数据水平拆分引入的问题主要是只能通过sharding key来读写操作，例如以userid为sharding key的切分例子，读userid的详细信息时，一定需要先知道userid,这样才能推算出再哪个cluster进而进行查询，假设我需要按username进行检索用户信息，需要引入额外的反向索引机制（类似HBASE二级索引），如在redis上存储username->userid的映射，以username查询的例子变成了先通过查询username->userid，再通过userid查询相应的信息。

实际上这个做法很简单，但是我们不要忽略了一个额外的隐患，那就是数据不一致的隐患。存储在redis里的username->userid和存储在mysql里的userid->username必须需要是一致

的，这个保证起来很多时候是一件比较困难的事情，举个例子来说，对于修改用户名这个场景，你需要同时修改redis和mysql,这两个东西是很难做到事务保证的,如mysql操作成功 但是redis却操作失败了（分布式事务引入成本较高）,对于互联网应用来说，可用性是最重要的，一致性是其次，所以能够容忍小量的不一致出现. 毕竟从占比来说，这类的不一致的比例可以微乎其微到忽略不计（一般写更新也会采用mq来保证直到成功为止才停止重试操作）

在这样的架构下，我们来看看数据存储的瓶颈是什么?

在这个拆分理念上搭建起来的架构，理论上不存在瓶颈（sharding key能确保各cluster流量相对均衡的前提下),不过确有一件恶心的事情，那就是cluster扩容的时候重做数据的成本，如我原来有3个cluster，但是现在我的数据增长比较快，我需要6个cluster，那么我们需要将每个cluster 一拆为二，一般的做法是

1.摘下一个slave,停同步,

2.对写记录增量log（实现上可以业务方对写操作 多一次写持久化mq 或者mysql主创建trigger记录写 等等方式）

3.开始对静态slave做数据, 一拆为二

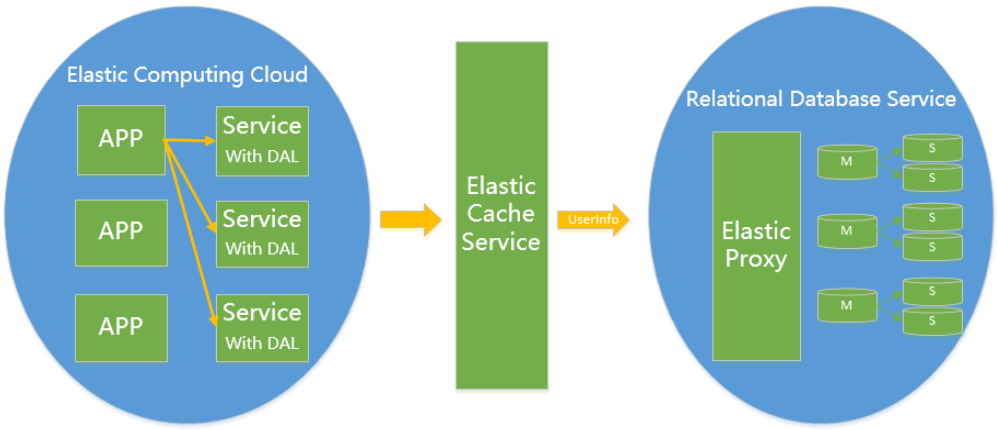
4.回放增量写入,直到追上的所有增量,与原cluster基本保持同步

5.写入切换，由原3 cluster 切换为6cluster

有没有类似飞机空中加油的感觉，这是一个脏活，累活，容易出问题的活，为了避免这个，我们一般在最开始的时候，设计足够多的sharding cluster来防止可能的cluster扩容这件事情

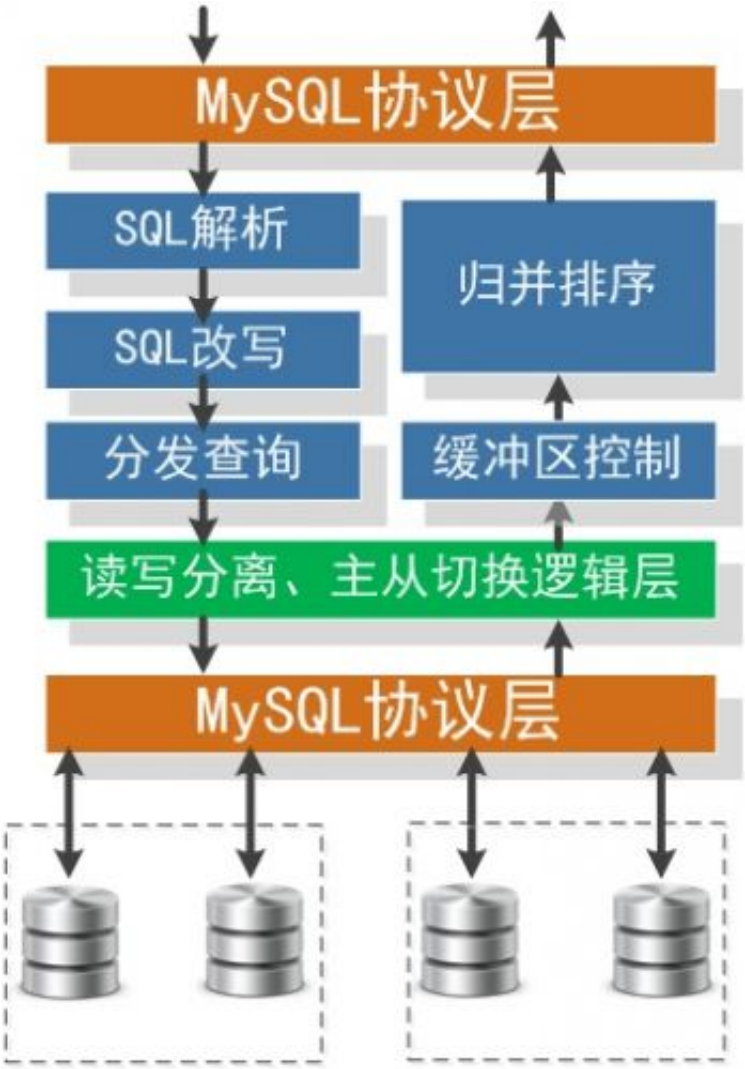
V5.0 云计算 腾飞（云数据库）

云计算现在是各大IT公司内部作为节约成本的一个突破口，对于数据存储的mysql来说，如何让其成为一个saas（Software as a Service）是关键点。在MS的官方文档中，把构建一个足够成熟的SAAS(MS简单列出了SAAS应用的4级成熟度)所面临的3个主要挑战：**可配置性，可扩展性，多用户存储结构设计称为"three headed monster"**. 可配置性和多用户存储结构设计在Mysql saas这个问题中并不是特别难办的一件事情，所以这里重点说一下可扩展性。Mysql作为一个saas服务，在架构演变为V4.0之后，依赖良好的sharding key设计, 已经不再存在扩展性问题，只是他在面对扩容缩容时，有一些脏活需要干，而作为saas,并不能避免扩容缩容这个问题，所以只要能把V4.0的脏活变成 **1. 扩容缩容对前端APP透明(业务代码不需要任何改动) 2.扩容缩容全自动化且对在线服务无影响** 那么他就拿到了作为Saas的门票.



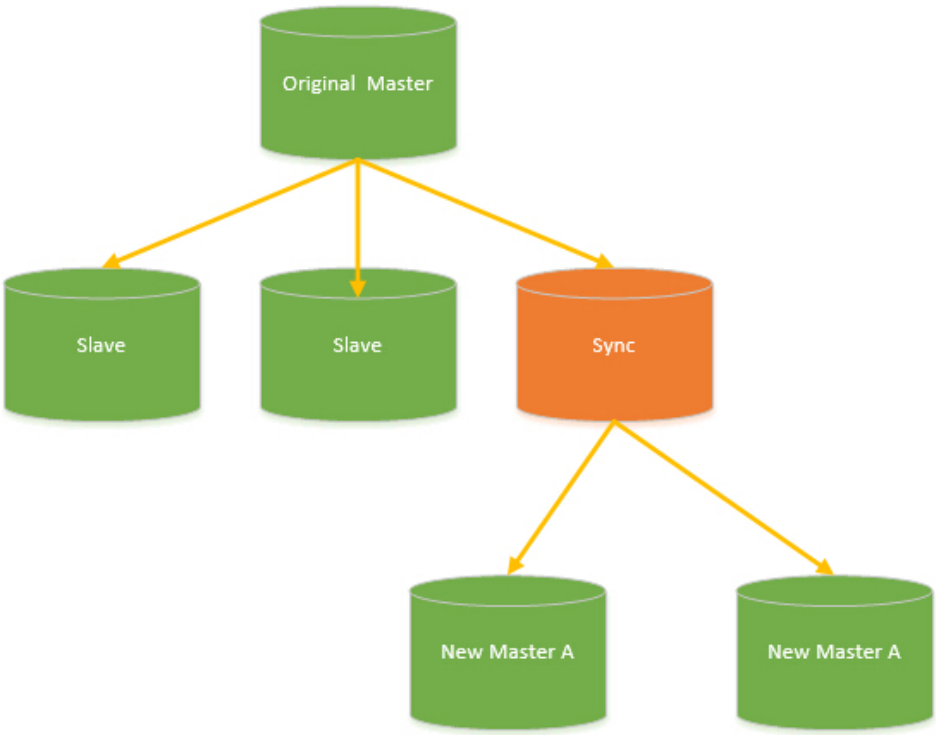
对于架构实现的关键点，**需要满足对业务透明，扩容缩容对业务不需要任何改动**，那么就必须eat our own dog food，在你mysql saas内部解决这个问题，一般的做法是我们需要引入一个Proxy,Proxy来解析sql协议，按sharding key 来寻找cluster, 判断是读操作还是写操作来请求主 或者 从，这一切内部的细节都由proxy来屏蔽。

这里借淘宝的图来列举一下proxy需要干哪些事情



公开的技术方案中也有类似的解决方案，见文章最后资料部分链接

对于架构实现的关键点，**扩容缩容全自动化且对在线服务无影响**； 扩容缩容对应到的数据操作即为数据拆分和数据合并，要做到完全自动化有非常多不同的实现方式，总体思路和V4.0介绍的瓶颈部分有关，目前来看这个问题比较好的方案就是实现一个伪装slave的sync slave，解析mysql同步协议，然后实现数据拆分逻辑，把全量数据进行拆分。具体架构见下图：



其中Sync slave对于Original Master来说，和一个普通的Mysql Slave没有任何区别，也不需要任何额外的区分对待。**需要扩容/缩容时，挂上一个Sync slave,开始全量同步+增量同步，等待一段时间追数据**。以扩容为例，若扩容后的服务和扩容前数据已经基本同步了，这时候如何做到切换对业务无影响？ 其实关键点还是在引入的proxy,这个问题转换为了如何让proxy做热切换后端的问题。这已经变成一个非常好处理的问题了.

另外值得关注的是：2014年5月28日——为了满足当下对Web及云应用需求，**甲骨文宣布推出MySQL Fabric**，在对应的资料部分我也放了很多Fabric的资料，有兴趣的可以看看，说不定会是以后的一个解决云数据库扩容缩容的手段

V more ?

等待革命...

其他资料

Dbproxy设计　http://tech.it168.com/a2012/0413/1337/000001337034.shtml
淘宝RDS 云数据库设计： http://blog.csdn.net/ywh147/article/details/8954625
http://www.infoq.com/cn/news/2012/10/taobao-ump
Mysql　Fabric
http://mysqlmusings.blogspot.jp/2013/09/brief-introduction-to-mysql-fabric.html
http://vnwrites.blogspot.jp/2013/09/mysqlfabric-sharding-introduction.html
http://vnwrites.blogspot.in/2013/09/mysqlfabric-sharding-example.html
http://vnwrites.blogspot.in/2013/09/mysqlfabric-sharding-migration.html
http://vnwrites.blogspot.jp/2013/09/mysqlfabric-sharding-maintenance.html