作为拥有世界上最多的互联网用户群体国家,尤其是移动互联网的大热，做到一个百万级的应用几乎是分分钟的事情。相应对技术的压力，和要求也是非常高的。

要应付这种大并发需要高性能系统的开发，先从常用的MySQL数据库碰到的性能瓶颈，来做分析。因为通常一个小项目刚开始一般都会只用mysql做为数据存储，当用户量增加的时候，就会出现数据库负载过高的问题，也就是所谓的慢查询。解决慢查询的问题一般来说，解决方案是优化SQL查询，读写分离和主从数据库，不断地切库分表。

1.SQL优化，最常见的方式是，优化联表查询，以及优化索引。这里面包括，尽量使用left join 替代 where联表；当碰到，频繁查询字段A和字段B，以及AB联合查询的情况时，对AB做联合索引，能够有效的降低索引存储空间，提升查询效率。在复杂联表的情况下，可以考虑使用 Memory中间表。

2.主从数据库和读写分离，主从分库是用来应对访问量增加，带来频繁读写导致数据库的访问和操作性能下降的问题。对数据库的操作，为了保证数据的完整性，通常涉及到锁的机制的问题。MySQL的InnoDB引擎支持行级锁，而MyIsAM只支持表锁定。这样的话，如果读写集中在一个表中的情况下，当访问量增加，就会造成明显的性能下降。因此，通过主从数据库的方式可以实现读写分离。一般来说，使用InnoDB来作为写库，使用MyISAM作为读库。这种方式是缺点当然是，数据库的维护难度增加，但是通常都会有专门的DBA这个职位来负责。而且几乎是必须的解决方案，算是基础设施了。

3.数据库分库和分表.有的时候会出现，某个表变得越来越庞大，比如存放message信息表，这样会造成读取性能的增加。这种情况下，你可以通过分表的方式来解决。将一个大表切分成若干个表。

一种简单算法是：

      设定表的大小为M，用户访问记录的ID和表的实际ID有差异，这个时候就需要做下换算，表的id = 用户访问的id 对M 进行求余;这种算法简单实现容易，单向扩展简单，但是缺点很明显，他是按照数量进行分配,但是往往实际情况会出现，访问量会集中在某几个表，而其他表访问不大，这样这种算法实际上实际上没太大的效果。而且重新调整的话，就比较困难。

类似分表的算法，还有很多，具体可以参考百度。

4.使用存储过程.将一些操作，直接通过存储过程的方式，预先设置在MySQL，客户端只需要调用存储过程就可以操作数据。在日常实践中，经常会出现，DBA在备份和恢复数据库的时候，遗忘了存储过程的情况。其次，业务调整的过程时，要对线上的存储过程进行调整，容易出现意想不到的问题，增加运维的成本。因此很多的团队不太愿意使用存储过程.

访问压力增大之后，最容易想到的解决方案就是，使用缓存了。实际上现实中，最常用的缓存无处不在。但是缓存细细说来其实还是比较复杂。

首先分为前端缓存，和后端缓存，两种技术解决方案。

先说前端页面的缓存，又可以分为PC端页面和移动客户端的缓存技术方案。

对PC端来说，对静态资源进行缓存，包括JS，CSS，图片等资源文件的缓存。

1. 一般来说，资源文件的缓存都是通过专门的CDN缓存加速，由于国内线路不通访问速度也不同，所以有专门的服务商提供专门的静态资源加速缓存服务保证不同的线路，以及用户可以就近快速访问。

2.对动态页面进行缓存，比如网站首页，内容页等这种改动不会太大的页面或者几乎不太会改变的页面，有两种方式：1）直接生成静态html页面，需要更新时通过后台成生成新页面进行覆盖。然后可以把静态页存放到本地硬盘或者同步到CDN上。2)使用vanish服务器作为方向代理，对php生成的动态页面进行缓存，由于vanish可以使用内存作为缓存，因此访问速度更快，且对生成页面的php代码不需要做任何的修改，就可以实现静态页面缓存。所以可以很好地解决因为一直遗留下来的问题导致代码修改成本高的情况。缺点也是，提升了运维成本。

对页面上某些内容经常变化的页面，比如用户中心页，其实也可以使用Ajax的方式来处理，将页面的基本内容缓存成静态页，使用ajax动态加载服务器端的动态数据。

浏览器缓存

通过Cache-Control,以及Last-Modified等控制缓存头的设置来告诉浏览器缓存页面。这样不必每次，都从服务器端重复请求新文件，也可以防止用户频繁刷新页面。

在过去带宽有限的情况，下浏览器缓存非常的重要，但是目前来说，由于带宽的提升，服务器性能提升，相对来说，这方面的浏览器的缓存要求相对下降了。使用浏览器缓存有的时候反而使得业务变得更加复杂，因此很多情况下，不少业务干脆不用浏览器缓存。

后端缓存

1.代码缓存：使用的各种PHP框架，本身会生成各种代码缓存，比如使用模板的视图文件，配置文件，都会解析成相应的php代码。通过这些方式，提升框架的运行效率。

2.数据缓存：这一类缓存实际上是最为复杂的一类的，也是经常会碰到，需要根据业务进行不断地调整一类的缓存。通常，我们常见的是对原先存储在MySQL数据的数据进行缓存加速。这也是前面涉及到的数据库性能瓶颈的通常解决方案之一。一般来说，过去常用的缓存服务器是memcached，但是随着Redis的出现之后，很多的创业公司，和项目，直接绕开memecached使用Redis做为缓存服务器，而且Redis不只是缓存服务，还有更多的高级特性。这里放在专门的段落来讨论，使用NoSQL替代MySQL的话题。

使用缓存最大的问题，是出现当缓存失效之后，如何解决惊群效应带来的服务器突然压力上升问题。当某个缓存失效之后，一般的做法是，再从后端的数据库中查找新的数据然后再重建缓存，但是在这个过程中，如果这个缓存内容同时有很多并发请求，就会出现，在重建新的缓存的时间段内，大量涌向后端数据库的访问，引起慢查询，导致数据库崩溃。一般来说，我们采取的方案是，主动更新缓存内容，同时延长缓存的时间，实际失效时间会比告知客户端的约定失效时间要长一些。比如实际失效时间是30s，约定失效时间是20s，后端的worker会对缓存进行主动更新，一般会使用两个key1,key2的方式进行轮流缓存和访问，比如，客户端访问的时候，先访问key1如果key1不存在则访问key2，缓存更新时，先生成key2，然后再删除key1。反之，亦然。

        再来说说所谓惊群效应，来源于一个很有意思的场景，在广场上，有一大群的鸽子，游人过来抛洒食物的时候，原本平静的广场，突然一大群鸽子都拥过来争抢游客手中的食物，当食物被吃完之后，又恢复了之前的平静。等到下一次食物到来的时候，又出现相同的情景。这种现象，我们叫做惊群效应,其实在生活中这种现象随处可见，比如：商场大促销，可以看见一大堆的人早已等候在门外，等商场开门的时候，大量的人如潮水一般涌入。还有线上的一些电商促销活动，比如淘宝双十一，各种秒杀活动等。都是常见的“惊群效应”现象。

        在技术开发中，我们经常也要考虑惊群效应出现的场景。

        我们最熟悉的一个业务场景就是，线上的秒杀促销活动的业务开发，在这个业务中就可以看到惊群效应的影响。我们可以假定一个业务场景，比如小米手机开放抢购。只有一万台手机，实际上参与秒杀的人，超过100万，假定时间是上午12点开始抢购，但是实际上真正的秒杀过程就是1两秒钟最多了，肯定被抢完了。

      惊群效应会导致，大量的服务器资源浪费，在服务器访问压力图表中，看到大多数情况下服务器是处于闲置状态的，一旦压力增加之后，服务器资源迅速被消耗殆尽，甚至导致崩溃，整个系统瘫痪。

       高并发状态下，惊群效应是经常出现，尤其是在基于社交的移动互联网产品中，几乎是家常便饭。我们经常处理的技术解决方案是，使用队列来进行处理，这也是NoSQL数据库常被用到的地方。将用户所有的请求，写入到队列中，然后通过后端的worker对队列的请求进行处理，这是一个生产者-消费者模型的经典使用场景。当处理完获取到规定数量级的结果之后，通知请求代理服务器，关闭请求通道，并重定向到别的页面，告知用户服务已经完结。比如，秒杀的时候，前端代理服务器负责将用户的请求发送到Redis队列服务器，然后后端的worker进程，消费队列的数据，当发现，秒杀的产品数量已经被抢光之后，则通知前端代理服务器，关闭秒杀请求的通道，重定向用户到一个静态提示页面，告知用户秒杀结束。这样可以保证，不会出现库存和订单不一致的情况，出现用户多抢的情况。包括抽奖也是一样，在高并发的情况下经常出现，用户抢到超出库存设定的相同商品。

      这里谈谈使用NoSQL在大型项目中的使用。

      这里说说我们常用的NoSQL开源项目，Redis，MongoDB，CounchBase等等，包括甚至一些新的语言，如，Node.JS等，这些新的技术产品，很多生来就是为，移动互联网大数据服务的。过去我们很多人都认为，所谓高并发，大数据，等名词都出现在BAT等少数公司里。但是随着移动互联网时代的到来，其实很多的新兴的移动互联网创业公司，很容易的，就能出现一夜间到上千万甚至过亿的用户，比如，之前在朋友圈风靡的，疯狂猜图，神经猫，这类小应用。所以在这些新兴的移动互联网产品中，NoSQL的使用几乎是基础服务。尤其是在社交类的应用中，对Feed数据的处理，消息推送，基于社交关系链的维护，用户社交行为的统计，都使得对代码的质量，数据的优化存储，业务的架构的设计等等都会有更高的要求。

NoSQL具体业务应用场景有以下几个方面：

1）队列服务。前面所提到的秒杀，抽奖，各种道具的交易等都会用到队列服务。其次是，消息推送，粉丝关注列表，等，都会用到队列服务。消息队列开源软件有很多，类似RabbitMQ，ZeroMQ,Redis等等，但是常用的还是Redis比较多。

对Feed的处理，是队列常用到的场景。比如微博的fee的消息，通常都是采用push和pull的策略，对活跃用户，一般通过缓存算法，LRC，LRF等进行计算，系统维护一个活跃用户的缓存池。然后为每一个活跃用户，生成他所关注的对象列表的消息队列，当用户上线的时候，主动推送这些消息队列。对不常活跃的用户，只有当他登录的时候，才会从后端的数据库去查询数据，生成结果。

这种方式可以有效的提升性能，同时节约存储空间。

2）计数器和限速器。移动社交应用中，用户的点赞行为非常常见，为了保存用户的点赞数据，我们经常会用到计数器服务，用被点赞的记录id作为key，对应的结果值为点赞次数。redis的incr,incrby等命令就经常被使用到，还可以通过expire来配合使用，在指定时间里面记录用户的数据。另外为了防止用户频繁访问API接口，尤其是恶意访问服务器数据服务，导致服务器过载，可以对用户的访问进行限速。

3）Top Rank，尤其是在游戏中经常会出现排名榜，电商网站也会有对商品的购买量进行排行，或者是会员等级排名等等。redis的ZSet经常被使用到，可以把记录的id存入zeset，设置对应的score为排名的依据的值，然后使用zRank来读取用户的排名，也可以使用zRange命令来获取到前指定位数的用户排行榜信息。

4）消息订阅。redis的pub/sub服务，可以实现消息订阅的功能，通常是用在用户的消息推送中，向指定用户推送相应的消息。其次是，任务分发，将用户产生的行为或者请求，发送到后端，在后端服务器中由后台worker进行处理，这样好处是，降低前端服务器的处理请求的压力，减少用户等待时间，同时减轻带宽的压力，也大大的降低服务器带宽成本，节约资金。

5）存储session。普通的网站使用file文件的方式存储session，使用redis作为session进行用户登录存储，好处很明显，一方面是访问性能提升，另外，redis使用set来存储在线用户的uid数据，这样可以很方便的使用sCard命令统计当前在线用户数，防止用户重登录。其次是避免，分布式环境下，带来用户登录访问不一致的问题。因为用户session信息，统一存储在独立的NoSQL数据库中一般来都是，使用redis服务器，这样，除了提升性能之外，也降低了代码处理的复杂度，甚至避免了过早使用大型的SSO系统，导致开发成本增加。

6）地理信息位置。MySQL其实也有存储地理位置信息，但是不如MongoDB，CounchBase等新兴数据库功能强大。基于地理位置的社交服务，用户会带着手机客户端，不断移动位置，所以用户的位置会不断地上报到服务器，然后根据服务器实时的地理信息位置，快速计算出附近附近的人，这对服务器的访问压力要求比较高。

通过代码，和各种软件的使用，来提升性能之外，整个系统架构的设计就提上日程了，最重要也是最通用的方式，自然是，堆服务器了，换个高大上的叫法，也就是所谓的分布式，系统集成。在高大上一点，就是所谓的云服务，各种所谓的云。

1）反向代理模式。所谓反向代理模式，用户所有的请求都统一发送到反向代理服务器，而反向代理服务器本身并不处理具体的请求，他只负责将请求发送到后端，获得结果之后，再传递给用户。这样的方式，能够加快整个系统的服务器响应，即使后端某个系统的崩溃，也不影响整个服务系统的奔溃。另外，这种反向代理服务器，一般来是应用服务器，本身也是分布式的，用户访问请求是被分发到不同的服务器上，这样能够支撑更大的并发请求。后端服务器，并不需要接入到互联网中，这样就不需要占用宝贵的带宽，只需要在本地网络使用本地线路进行连接，专注各种数据的运算服务，这样能够大大的降低成本，提高系统的安全性，同时提升整个系统的性能。

2）过滤器中间件。很多的请求并非都是立即被执行，或者是都会被请求。并非所有的客户的无理要求都必须满足一样，有的餐厅就并不一定提供住宿服务一样。当用户请求，被传递到中间层中，会有各种中间件会对用户的请求进行过滤，比如秒杀或者抽奖时，用户的频繁刷新请求，会被过滤器进行过滤，合并或者拒绝用户的请求。这样真正传递到后端的请求就会减少很多，有效地降低了后端服务器的压力。

3）数据视图。用户的很多重要数据都最终存在数据库中，但是用户的每次请求，并不需要一次性访问所有的数据。比如，判定用户是否登录，只需要访问用户的是否登陆的标记。这些数据可以存在redis的key中，并不需要通过数据库去保存。再有，一些信息页的数据，由于变动不会太大，由后台编辑生成，而并非是用户生成的信息数据，往往是树状结构，每次通过数据库查询，需要大量重复的join或者多次的sql查询才能读取。一方面增加了，代码的逻辑设计和维护的成本，另外一方面，运行成本也大大提升，即使通过缓存的方式，也会有惊群效应的问题需要解决。所以不如通过主动视图的方式来解决，这一类的数据，可以通过使用后台使用mysql进行存储和编辑，而面向用户的前台数据，使用MongoDB等高性能的NoSQL的数据库存储，相当于，MongoDB作为MySQL的视图。

4）缓存策略。

对于用户产生的UGC内容，它的特点是，有的用户行为读写频繁，其次是往往出现不可预测，突然一下子并发提升。比如，用户的内容分享，转发，收藏，点赞。有的帖子，会突然出现被病毒式传播的现象，大量的用户分享，转发。可能这篇文章的访问量，就会突然增加，相应的访问压力也提升，对数据库的读取压力也增加。

然而并非所有的帖子都是热门帖子，都会被访问，往往在一个系统中，大部分的帖子都处于冷门或者无人问津的状态下，只有少部分的帖子汇出现热门的状态。

因此并非所有的内容都应该被缓存，提供统一的，缓存策略服务就显得很有必要。缓存策略服务器系统，使用不同的算法策略，维持一个缓存数据的池，并非是简单地缓存数据。使用有效地缓存策略，对低访问压力的请求直接穿透缓存使用后端访问，高访问压力使用缓存服务。

缓存策略的算法有很多种，常见的有：

Least Frequently Used（LFU）

Least Recently User（LRU）

Least Recently Used 2（LRU2）

Two Queues（2Q）

具体还可以百度查询，翻阅更多的详细介绍。

5）任务系统。将很多的耗费性能的服务，集成为一个专门的服务系统，比如，定时任务服务，图像处理，分布式文件同步等等。