推荐引擎 <https://www.ibm.com/developerworks/cn/web/1103_zhaoct_recommstudy1/>

推荐引擎利用特殊的信息过滤技术，将不同的物品或内容推荐给可能对它们感兴趣的用户.显式的用户反馈（Explicit Feedback）能准确的反应用户对物品的真实喜好，但需要用户付出额外的代价，而隐式的用户行为(Implicit Feedback)，通过一些分析和处理，也能反映用户的喜好，只是数据不是很精确，有些行为的分析存在较大的噪音

如何发现数据的相关性(Relevance)：

1.根据系统用户的基本信息发现用户的相关程度，这种被称为基于人口统计学的推荐（Demographic-based Recommendation）

2.根据推荐物品或内容的元数据，发现物品或者内容的相关性，这种被称为基于内容的推荐（Content-based Recommendation）

3.根据用户对物品或者信息的偏好，发现物品或者内容本身的相关性，或者是发现用户的相关性，这种被称为基于协同过滤的推荐（Collaborative Filtering-based Recommendation）

基于用户的协同过滤推荐机制和基于人口统计学的推荐机制都是计算用户的相似度，并基于“邻居”用户群计算推荐，但它们所不同的是如何计算用户的相似度，基于人口统计学的机制只考虑用户本身的特征，而基于用户的协同过滤机制可是在用户的历史偏好的数据上计算用户的相似度，它的基本假设是，喜欢类似物品的用户可能有相同或者相似的口味和偏好

How to recommend 推荐策略:

1.基于物品和用户本身的，这种推荐引擎将每个用户和每个物品都当作独立的实体，预测每个用户对于每个物品的喜好程度，这些信息往往是用一个二维矩阵描述的。由于用户感兴趣的物品远远小于总物品的数目，这样的模型导致大量的数据空置，即我们得到的二维矩阵往往是一个很大的稀疏矩阵。同时为了减小计算量，我们可以对物品和用户进行聚类， 然后记录和计算**一类用户对一类物品**的喜好程度，但这样的模型又会在推荐的准确性上有损失

2.基于关联规则的推荐（Rule-based Recommendation）：关联规则的挖掘已经是数据挖掘中的一个经典的问题，主要是挖掘一些数据的依赖关系，典型的场景就是“购物篮问题”，通过关联规则的挖掘，我们可以找到**哪些物品经常被同时购买**，或者用户购买了一些物品后通常会购买哪些其他的物品，当我们挖掘出这些关联规则之后，我们可以基于这些规则给用户进行推荐

3.基于模型的推荐（Model-based Recommendation）：这是一个典型的机器学习的问题，可以将**已有的用户喜好信息作为训练样本**，训练出一个预测用户喜好的模型，这样以后用户在进入系统，可以基于此模型计算推荐。这种方法的问题在于如何将用户**实时**或者近期的喜好信息反馈给训练好的模型，从而提高推荐的准确度

其实在现在的推荐系统中，很少有只使用了一个推荐策略的推荐引擎，一般都是在不同的场景下使用不同的推荐策略从而达到最好的推荐效果，例如 Amazon 的推荐，它将基于用户本身历史购买数据的推荐，和基于用户当前浏览的物品的推荐，以及基于大众喜好的当下比较流行的物品都在不同的区域推荐给用户，让用户可以从全方位的推荐中找到自己真正感兴趣的物品

推荐策略 in Amazon - 分区的混合（Mixed Hybridization）:

Amazon 利用可以记录的所有用户在站点上的行为，根据不同数据的特点对它们进行处理，并分成不同区为用户推送推荐：

今日推荐 (Today's Recommendation For You): 通常是根据用户的近期的历史购买或者查看记录，并结合时下流行的物品给出一个折中的推荐。

新产品的推荐 (New For You): 采用了基于内容的推荐机制 (Content-based Recommendation)，将一些新到物品推荐给用户。在方法选择上由于新物品没有大量的用户喜好信息，所以基于内容的推荐能很好的解决这个“冷启动”的问题。

捆绑销售 (Frequently Bought Together): 采用数据挖掘技术对用户的购买行为进行分析，找到经常被一起或同一个人购买的物品集，进行捆绑销售，这是一种典型的基于项目的协同过滤推荐机制。

别人购买 / 浏览的商品 (Customers Who Bought/See This Item Also Bought/See): 这也是一个典型的基于项目的协同过滤推荐的应用，通过社会化机制用户能更快更方便的找到自己感兴趣的物品

倒排索引：<http://blog.csdn.net/hguisu/article/details/7962350>

倒排索引(Inverted Index)：倒排索引是实现“单词-文档矩阵”的一种具体存储形式，通过倒排索引，可以根据单词快速获取包含这个单词的文档列表。倒排索引主要由两个部分组成：“单词词典”和“倒排文件”

有了这个索引系统，搜索引擎可以很方便地响应用户的查询，比如用户输入查询词“Facebook”，搜索系统查找倒排索引，从中可以读出包含这个单词的**文档**，这些文档就是提供给用户的搜索结果，而利用**单词频率信息**、文档频率信息即可以对这些候选搜索结果进行排序，计算文档和查询的相似性，按照相似性得分由高到低排序输出，此即为搜索系统的部分内部流程

单词词典是倒排索引中非常重要的组成部分，它用来维护文档集合中出现过的所有单词的相关信息，同时用来记载某个单词对应的倒排列表在倒排文件中的位置信息。在支持搜索时，根据用户的查询词，去单词词典里查询，就能够获得相应的倒排列表，并以此作为后续排序的基础。对于一个规模很大的文档集合来说，可能包含几十万甚至上百万的不同单词，能否快速定位某个单词，这直接影响搜索时的响应速度，所以需要高效的数据结构来对单词词典进行构建和查找，常用的数据结构包括哈希加链表结构（HashMap）和树形词典结构(Trie, B+ Tree)

When to use MongoDB or other document oriented database systems? <https://www.zhihu.com/question/20059632>

如果数据集比较简单，没有Join，MongoDB会非常棒。没错。但这不意味着如果数据复杂就不行。相反，这时需要重新设计数据schema (模式、架构、结构)，把相关的内容放到一个document里，这是与关系型数据库追求的三范式最不一样的地方。比如一篇文章的评论不算太多，最多几百个，就可以把评论放到数组(array)里，作为文章这一document的一部分。是的，数组(array)是支持的，查询数组元素也非常自然。这样的好处是在读取文章这一document时，一个页面上所有需要的数据都有了，读硬盘（内存）的次数少了，自然就快，相比之下，关系型数据库Join就麻烦很多了。这个例子里数据的access pattern决定了数据schema

1. Document-oriented.比如电子商务网站的产品除了预设的价格，id等，因具体产品不同有许多自定义属性，产品这个表应该怎么设计？就算一类产品大致相似，也有独特的属性，比如LCD和LED显示器的产品特性就不一样。这个问题有一些解决方案，比如 Entity-attribute-value Model 但是这些方案都很复杂，这时，schemaless的优点就显现出来了。
2. Easy scalability. 当数据规模大到一个机器装不下了，或者一台机器数据读写负载太高，需要使用cluster的时候，关系型数据库的partitioning, sharding要复杂的手工处理。MongoDB可以自动按照用户给定的sharding key把数据分片，并且动态地平衡各个机器的数据量。
3. Rich query language. 这是关系型数据库擅长的地方，也是用户所希望的，而有的key-value数据库只把value当作一个binary blob，比如Voldemort，只能通过key查询。简言之，不通用。MongoDB内部也是按key-value存的，但是支持各种查询，并且可以建各种索引，提供了易用与高性能。

实现一个数据库最难的地方就是如何在保证并发访问的情况下，最大量的把随机IO转化为顺序IO，这也是我不太看好很多NOSQL的原因，你就实现了一个人家的功能的子集，然后跑出来说，哦，我好牛逼哦，我快的多。这不是笑话是什么。还有那种动不动就说比rdbms快几万倍的数据库系统，我只是想问你保证事务么，我需要事务支持，你支持行级锁么

The Log: What every software engineer should know about real-time data's unifying abstraction <https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying>

A log is perhaps the simplest possible storage abstraction. It is an append-only, totally-ordered sequence of records ordered by time. So, a log is not all that different from a file or a table. A file is an array of bytes, a table is an array of records, and a log is really just a kind of table or file where the records are sorted by time.

Logs in databases. The log is the record of what happened, and each table or index is a projection of this history into some useful data structure or index. Since the log is immediately persisted it is used as the authoritative source in restoring all other persistent structures in the event of a crash.Over-time the usage of the log grew from an implementation detail of ACID to a method for replicating data between databases. It turns out that the sequence of changes that happened on the database is exactly what is needed to keep a remote replica database in sync.

The log also acts as a buffer that makes data production asynchronous from data consumption. This is important for a lot of reasons, but particularly when there are multiple subscribers that may consume at different rates(point of time).

GeoHashing <http://www.cnblogs.com/subaiBlog/p/5410590.html>

GeoHash算法能够将坐标变成特定的编码，然后进行对应Hash，还能够根据编码的前缀，来进行判断两点是否在附近：

1.将地图四分，也就是分成左上、右上、左下、右下四个部分，然后对应的地图块的编码后面追加"01","11","00","10"

|01|11|

|00|10|

2.然后再将四分后的各个地图块，重复步骤1，不断地进行四分，编码也会两位两位地进行增加。直到地图块不能再进行四分。

3.得到对应的每个坐标的编码。

因为是按着每个地图4分的，所以这地图四分前的编码是相同的，即使4分之后，这4块地图还是有着相同的前缀，因此，我们可以根据编码的最长相同前缀，去找出距离最近的加油站的坐标.

GeoHash一般会产生两种误差

1.在边界附近查询的时候，明明在附近但是因为跨边界，导致前缀根本就不一样 这个的解决方法是，除了使用定位点的GeoHash编码进行匹配外，还使用周围8个区域的GeoHash编码，这样可以避免这个问题

2.前缀差相近点可能距离很远，解决方法是最后计算一遍实际距离。

在实际的GeoHash的计算过程中，我们往往会做一些调整

1 2 3

4 5 6

7 8 9

比如我们利用prefix求出5这个区域这个距离内的所有点，实际上我们还可以对这个区域相邻的周围的1，2，3，4，6，7，8，9做一个扫描。

也就是我们仍然可以通过计算得出周围八个区域的prefix，然后去查询。

最后对于所有点，重新计算一下真实具体，找出真正的 < 3km 的点。

用户密码的存储：<http://zhuoqiang.me/password-storage-and-python-example.html>

密码最好是以不可还原明文的方式来保存。通常利用哈希算法的单向性来保证明文以不可还原的有损方式进行存储

1.使用自己独创的哈希算法对密码进行哈希，存储哈希过的值.哈希算法复杂，独创对理论要求很高。一般独创的哈希算法肯定没有公开经过时间检验的算法质量高，天才另算

2.使用 MD5 或 SHA-1 哈希算法. MD5 和 SHA-1 已破解。虽不能还原明文，但很容易找到能生成相同哈希值的替代明文。而且这两个算法速度较快，暴力破解相对省时，建议不要使用它们。任何情况下尽可能的不要使用md5算法，而使用SHA系列的哈希算法。因为md5算法在很多地方被证明是很容易冲突的

3.使用更安全的 SHA-256 等成熟算法. 更加复杂的算法增加了暴力破解的难度。但如果遇到简单密码，用彩虹字典的暴力破解法，很快就能得到密码原文. 单向Hash算法（MD5, SHA1, SHA256等）可以保证管理员几乎不能恢复原始密码。但它有两个特点： 1）从同一个密码进行单向哈希，得到的总是唯一确定的摘要 2）计算速度快。随着技术进步，尤其是显卡在高性能计算中的普及，一秒钟能够完成数十亿次单向哈希计算. 结合上面两个特点，考虑到多数人所使用的密码为常见的组合，攻击者可以将所有密码的常见组合进行单向哈希，得到一个摘要组合, 然后与数据库中的摘要进行比对即可获得对应的密码。这个摘要组合也被称为rainbow table

4.加入随机 salt 的哈希算法.密码原文（或经过 hash 后的值）和随机生成的 salt 字符串混淆，然后再进行 hash，最后把 hash 值和 salt 值一起存储。验证密码的时候只要用 salt 再与密码原文做一次相同步骤的运算，比较结果与存储的 hash 值就可以了。这样一来哪怕是简单的密码，在进过 salt 混淆后产生的也是很不常见的字符串，根本不会出现在彩虹字典中。salt 越长暴力破解的难度越大. 具体的 hash 过程也可以进行若干次叠代，虽然 hash 叠代会增加碰撞率，但也增加暴力破解的资源消耗。就算真被破解了，黑客掌握的也只是这个随机 salt 混淆过的密码，用户原始密码依然安全，不用担心其它使用相同密码的应用. 将明文密码混入“随机因素“，然后进行单向哈希后存储，也就是所谓的”Salted Hash”。 这个方式相比上面的方案，最大的好处是针对每一个数据库中的密码，都需要建立一个完整的rainbow table进行匹配。因为两个同样使用“passwordhunter”作为密码的账户，在数据库中存储的摘要完全不同.

上面这几种方法都不可能得到密码的明文，就算是系统管理员也没办法。对于那些真的忘了密码的用户，网站只能提供重置密码的功能了. 绝对不能存可还原密码原文的信息

如果因为种种原因一定要可还原密码原文，请使用非对称加密，并保管好私钥. 密码经过非对称加密后再存储. 密码的安全性等同于私钥的安全性。密码明文经过公钥加密。要还原密码明文，必须要相应的私钥才行。因此只要保证私钥的安全，密码明文就安全。私钥可以由某个受信任的人或机构来掌管，身份验证只需要用公钥就可以了. 实际上，这也是 HTTPS/SSL 的理论基础。这里的关键是 私钥的安全 ，如果私钥泄露，那密码明文就危险了。

Session: <http://fred-zone.blogspot.com/2014/01/web-session.html>

Session 之所以会存在，是因为HTTP 为stateless 的设计，Server 和Client 不会一直保持连线状态，也不会有双方状态的即时更新，所以，Server 并不知道Client 的状态（像是是否已经登入）。因此，后来的网站开发者，采用Session 这样的设计来解决这问题.

在最原始的Session 设计，大多开发者都将资料存在Server 上，也就是你点了什么饮料，都是记录在Server 里，可能是Database、记忆体或是档案，可以以任何一种形式储存。然后，当你去领饮料时，店员会输入你的号码，用你的号码得知你是否点过餐、点了什么东西。

一般的小网站，这样的解决方案并没有什么问题。但是对今天这种超大流量的网站服务来说，因为他们有无数台对外的Server，有如无数个服务窗口，让顾客总是随机进入其中一个窗口来兑换饮料，所以后端怎么存放和共享这个session资料，又要兼顾效能和方便维护，就变成是很大的问题。

因此Cookie-based Session 就被提出为一个解决方案，把资料暂存放在Cookie 中，让Client 自己负责保存。简单来说，就是把你点什么饮料，通通直接写在号码牌上。Server 就可以直接看你的号码牌上写了什么，而不必花大量时间去后面建立大规模的Server 来处理Session. 不过，这边要特别提到，因为cookie 有4K 资料大小的限制，很多网站服务会选择cookie-based 和后端储存并行的方案.或许有人会问到Cookie 是否有可能被篡改？这类安全问题，通常会使用加密手段来解决。一般来说，Cookie-based session 的cookie 会被加密，只有Server 才知道如何解开，Client 并没有能力可以存取，只是得到一个看不懂的包裹，所以不会有安全性的问题（当然还是有机会被破）. 由于仍然存在风险，这也是其中一个原因，为什么有些网站仍然会采用cookie-based 和后端储存Session 并行的解决方案，或是会避免把敏感资料放在Cookie-based Session 上.

拿号码牌去Server 要资料，主要也分为两种方法，Cookie (如上所述)和运用标准的Query string/POST body方法。（其实只要能把号码传到Server 上，任何方法都行）. 时常会听到『使用Session 传值』这类说法，其实就是利用Session 机制储存资料，让不同页面之间可以互相传递资料。其原理通常是使用Query String 或POST body 等方法，把资料往Server 传之后，在Server 端将Client 上传的资料存在Session 之中。之后的连线或开启其它页面时，因为你拿的号码牌是同一个，所以在不同的页面之下，仍然可以读到前一次所储存在Session 的状态。

URL地址重写是对客户端不支持Cookie的解决方案。URL地址重写的原理是将该用户Session的id信息重写到URL地址中。服务器能够解析重写后的URL获取Session的id。这样即使客户端不支持Cookie，也可以使用Session来记录用户状态

Cookie是存在客户端， Session是存在服务端内存。由于会有越来越多的用户访问服务器，因此Session也会越来越多。为防止内存溢出，服务器会把长时间内没有活跃的Session从内存删除。这个时间就是Session的超时时间。如果超过了超时时间没访问过服务器，Session就自动失效了。 By default, 用户关了浏览器 session就失效

ACID, CAP:　<http://in355hz.iteye.com/blog/2029963>

强一致性(Consistency)。系统在执行过某项操作后仍然处于一致的状态。在分布式系统中，更新操作执行成功后所有的用户都应该读取到最新的值，这样的系统被认为具有强一致性。

可用性(Availability)。每一个操作总是能够在一定的时间内返回结果，这里需要注意的是“一定时间内”和“返回结果”。

分区容错性(Partition Tolerance)。分区容错性可以理解为系统在存在网络分区的情况下仍然可以接受请求(满足一致性和可用性)。这里网络分区是指由于某种原因网络被分成若干个孤立的区域，而区域之间互不相通。还有一些人将分区容错性理解为系统对节点动态加入和离开的处理能力，因为节点的加入和离开可以认为是集群内部的网络分区。

如果Partition-tolerance的需求为0，那么则CA， 延伸出ACID协议， 譬如关系型DBMS系统的事务一致性。然而对于互联网的分布式系统，鉴于高压力、大数据，那么必须对分区容忍性要求高，且高可用性，则采用BASE 弱一致性或者最终一致性。而对于很多互联网应用来说完全牺牲一致性是不可取的，否则数据是混乱的，那么系统可用性再高分布式再好也没有了价值。牺牲一致性，只是不再要求关系型数据库中的强一致性。 从客户体验出发，最终一致性的关键是时间窗口，尽量让用户“用户感知到的一致性”

ACID是事务所具有的特性。数据库实现 ACID 最关键的技术是**日志**和**锁**

原子性（Atomicity）：事务中的操作要么都做，要么都不做。数据库依赖 Redo / undo 日志实现事务的原子性

一致性（Consistency）：系统必须始终处在强一致状态下。用类型检查、唯一索引、外键约束和级联更新保护数据的完整性

隔离性（Isolation）：一个事务的执行不能被其他事务所干扰。实现事务隔离的主要手段是锁

持续性（Durability）：一个已提交的事务对数据库中数据的改变是永久性的。数据库的持久化依赖磁盘和数据复制机制

保证ACID特性是传统关系型数据库中事务管理的重要任务，也是恢复和并发控制的基本单位。

一致性模型：<http://coolshell.cn/articles/10910.html>

Balance between Availability, Consistency, Latency

1）Weak 弱一致性：当你写入一个新值后，读操作在数据副本上可能读出来，也可能读不出来。比如：某些cache系统，网络游戏其它玩家的数据和你没什么关系。

2）Eventually 最终一致性：当你写入一个新值后，有可能读不出来，但在某个时间窗口之后保证最终能读出来。比如：DNS，电子邮件、Amazon S3，Google搜索引擎这样的系统。

3）Strong 强一致性：新的数据一旦写入，在任意副本任意时刻都能读到新值。

首先是**Master-Slave**结构，对于这种结构，Slave一般是Master的备份。在这样的系统中，一般是如下设计的：

1）读写请求都由Master负责。

2）写请求写到Master上后，由Master同步到Slave上。

从Master同步到Slave上，你可以使用异步，也可以使用同步，可以使用Master来push，也可以使用Slave来pull。 通常来说是Slave来周期性的pull，所以，是最终一致性。这个设计的问题是，如果Master在pull周期内垮掉了，那么会导致这个时间片内的数据丢失。如果你不想让数据丢掉，Slave只能成为Read-Only的方式等Master恢复。当然，如果你可以容忍数据丢掉的话，你可以马上让Slave代替Master工作（对于只负责计算的结点来说，没有数据一致性和数据丢失的问题，Master-Slave的方式就可以解决单点问题了） 当然，Master Slave也可以是强一致性的， 比如：当我们写Master的时候，Master负责先写自己，等成功后，再写Slave，两者都成功后返回成功，整个过程是同步的，如果写Slave失败了，那么两种方法，一种是标记Slave不可用报错并继续服务（等Slave恢复后同步Master的数据，可以有多个Slave，这样少一个，还有备份，就像前面说的写三份那样），另一种是回滚自己并返回写失败

**Master-Master**，又叫Multi-master，是指一个系统存在两个或多个Master，每个Master都提供read-write服务。这个模型是Master-Slave的加强版，数据间同步一般是通过Master间的异步完成，所以是最终一致性。 Master-Master的好处是，一台Master挂了，别的Master可以正常做读写服务，他和Master-Slave一样，当数据没有被复制到别的Master上时，数据会丢失。很多数据库都支持Master-Master的Replication的机制.另外，如果多个Master对同一个数据进行修改的时候，这个模型的恶梦就出现了——对数据间的冲突合并，这并不是一件容易的事情。看看Dynamo的Vector Clock的设计（记录数据的版本号和修改者）就知道这个事并不那么简单，而且Dynamo对数据冲突这个事是交给用户自己搞的。就像我们的SVN源码冲突一样，对于同一行代码的冲突，只能交给开发者自己来处理

**Two/Three Phase Commit 2PC**说白了就是第一阶段做Vote，第二阶段做决定的一个算法，也可以看到2PC这个事是强一致性的算法

这个协议的缩写又叫2PC，中文叫两阶段提交。在分布式系统中，每个节点虽然可以知晓自己的操作时成功或者失败，却无法知道其他节点的操作的成功或失败。当一个事务跨越多个节点时，为了保持事务的ACID特性，需要引入一个作为**协调者Coordinator**的组件来统一掌控所有节点(称作**参与者 Cohort**)的操作结果并最终指示这些节点是否要把操作结果进行真正的提交(比如将更新后的数据写入磁盘等等)。 两阶段提交的算法如下：

第一阶段：

协调者会问所有的参与者结点，是否可以执行提交操作。

各个参与者开始事务执行的准备工作：如：为资源上锁，预留资源，写undo/redo log……

参与者响应协调者，如果事务的准备工作成功，则回应“可以提交”，否则回应“拒绝提交”。

第二阶段：

如果所有的参与者都回应“可以提交”，那么，协调者向所有的参与者发送“正式提交”的命令。参与者完成正式提交，并释放所有资源，然后回应“完成”，协调者收集各结点的“完成”回应后结束这个Global Transaction。

如果有一个参与者回应“拒绝提交”，那么，协调者向所有的参与者发送“回滚操作”，并释放所有资源，然后回应“回滚完成”，协调者收集各结点的“回滚”回应后，取消这个Global Transaction。

三段提交3PC的核心理念是：在询问的时候并不锁定资源，除非所有人都同意了，才开始锁资源。

从B 树、B+ 树、B\* 树谈到R 树 <http://blog.csdn.net/v_JULY_v/article/details/6530142>

动态查找树主要有：二叉查找树（Binary Search Tree），平衡二叉查找树（Balanced Binary Search Tree），红黑树(Red-Black Tree )，B-tree/B+-tree/ B\*-tree (B~Tree)。前三者是典型的二叉查找树结构，其查找的时间复杂度O(log2N)与树的深度相关，那么降低树的深度自然会提高查找效率

若B树某一非叶子节点包含N个关键字，则此非叶子节点含有N+1个孩子结点.

B+ Tree：是应文件系统所需而产生的一种B-tree的变形树。

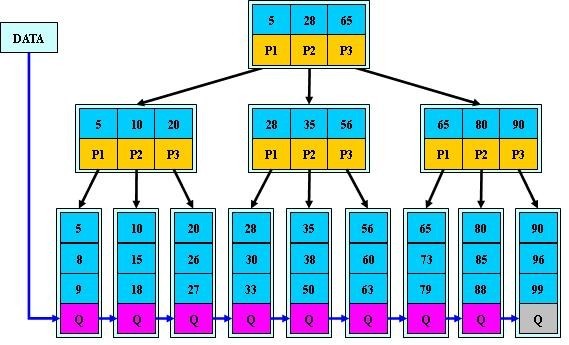
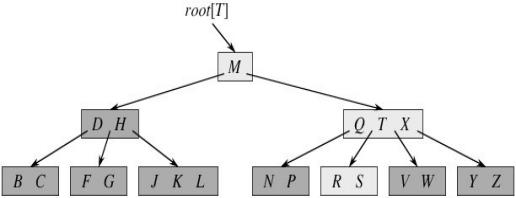
一棵m阶的B+树和m阶的B树的异同点在于：

1.有n棵子树的结点中含有n 个关键字

2.所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接。 (而B 树的叶子节点并没有包括全部需要查找的信息)

3.所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键字。 (而B 树的非终节点也包含需要查找的有效信息)

Here are B-Tree and B+ Tree:



B+树还有一个最大的好处，方便扫库，B树必须用中序遍历的方法按序扫库，而B+树直接从叶子结点挨个扫一遍就完了，B+树支持range-query非常方便，而B树不支持。这是数据库选用B+树的最主要原因. 比如要查 5-10之间的，B+树一把到5这个标记，再一把到10，然后串起来就行了，B树就非常麻烦。B树的好处，就是成功查询特别有利，因为树的高度总体要比B+树矮。不成功的情况下，B树也比B+树稍稍占一点点便宜。

R树是一种能够有效进行高维空间搜索的数据结构.

多线程同步互斥-条件变量与信号量,生产者与消费者问题 : <http://www.cnblogs.com/lcw/p/3236602.html>

条件变量被用来阻塞一个线程，当条件不满足时，线程往往解开相应的互斥锁并等待条件发生变化。一旦其它的某个线程改变了条件变量，它将通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。这些线程将重新锁定互斥锁并重新测试条件是否满足. 一般来说，条件变量被用来进行线程间的同步。

生产者消费者问题（Producer-consumer problem），也称有限缓冲问题（Bounded-buffer problem），是一个多线程同步问题的经典案例。该问题描述了共享固定大小缓冲区的两个线程——即所谓的“生产者”和“消费者”——在实际运行时会发生的问题。生产者的主要作用是生成一定量的数据放到缓冲区中，然后重复此过程。与此同时，消费者也在缓冲区消耗这些数据。该问题的关键就是要保证生产者不会在缓冲区满时加入数据，消费者也不会在缓冲区中空时消耗数据

Design 大规模，高并发的订票系统 <http://itindex.net/detail/39902-%E7%B3%BB%E7%BB%9F-%E8%AE%BE%E8%AE%A1>

1. 车次和剩余票量的查询。

考虑到车次查询量可能是订单数量的数倍至数十倍，不能让用户提交查询请求时直接去主数据库检索数据，而应该采用**前端+缓存+检索+数据库**的多层逻辑结构。数据库存放持久化的权威数据并保证数据的一致性；缓存层负责把车次、余量等数据放到内存中以保证最好的查询性能，并有比较好的横向扩展性；检索机负责定时（例如每5秒一次）去数据库检索所有车次信息并主动更新缓存机上的数据；前端负责响应来自用户的web请求。这个架构无法保证用户看到的车票余量是实时准确的（比如有数秒的滞后），但由于用户从看到车票余量到完成订单之间肯定是有时间间隔的，在订票高峰期和票量较少时本来就无法保证“在看到有票的情况下一定能订到票”（技术上能够实现这一点，但代价非常大），所以这个缺陷并不明显，是个很划得来的折中。注意是检索机负责将车票数据抓出来并更新到缓存机上，这是保护数据库并使缓存层能够线性扩展的关键方法。另外查询页面需要采取防频繁刷新的措施，这个在前端机上设置web server策略即可。

1. 下单部分由于要更新车票余量，必须保证数据的一致性，扩展性不可能很好，因此是整个系统中最为脆弱的一环。

实现的方法分同步处理和异步处理两种。**同步处理**就是用户选择完车次正式下单订票后，立即锁住车票记录并检查车票真实余量，如果大于1，那么余量减1，解除锁定并回复用户订票成功进入支付流程，否则解除锁定回复订票失败请用户选择其它车次。这是订票系统的标准流程，无论用户量大还是小，处理流程都是一样的。为了支撑春运这种极端情况下的高访问量，需要提高订单处理的并发吞吐量和单个事务的处理速度。提高吞吐量可以将不同车次的车票数据分拆到不同的物理服务器上，提高订单处理速度可以考虑取消关系数据库，将车次数据放到内存中并用原生语言实现订单处理逻辑。有一个比较值得考虑的措施是在用户下单前用图片或者短信的方式要求用户二次验证，这既可以防止刷页面，也可以使峰值变得更平缓。**异步处理**就是在用户提交订单时并不立即告诉用户订票成功或者失败，只是将订票请求放入队列里排队，订单成功处理后再通知用户。处理优先级上采用时间排序或者抽签都可以，不过抽签适合在非常时期采用，并不适合作为一个标准策略，这多少增加了系统开发的复杂度。采用异步的方式将会在最大程度上避免用户下单高峰造成的冲击，缺点是用户不知道什么时候能有结果，是否应该尝试其它车次，这对用户体验有一定程度的损伤

<http://coolshell.cn/articles/6470.html>

抢票与秒杀系统的比较：抢票，一方面会伴随着大量的查询操作，更BT的是下单的时候需要对数据库很多的一致性的操作，一方面是从起点到终点各个分段票的一致性，另一方面，买的人路线、车次、时间选择有很多，会不停地改变下单方式。而秒杀，直接杀就好了，没有那么多查询和一致性的问题。另外，关于秒杀，完全可以做成只接受前N个用户的请求（完全不操作后端的任何数据， 仅仅只是对用户的下单操作log），这种业务，只需要在内存cache中放好可秒杀的数量，还可以把数据分布开来放，100商品，10台服务器一台放10个，无需在当时操作任何数据库。可以订单数够后，停止秒杀，然后批量写数据库。而且秒杀的商品不多。火车票这个不是像秒杀那么简单的，春运时间，几乎所有的票都是热门票，而且几乎是全国人民都来了，而且还有转车业务，多条线的库存都要做事务操作

订票系统应该和电子商务的订单系统很相似，都是需要对库存进行：1）占住库存，2）支付，3）扣除库存的操作。这个是需要有一致性的检查的，也就是在并发时需要对数据加锁的。B2C的电商基本上都会把这个事干成异步的，也就是说，你下的订单并不是马上处理的，而是延时处理的，只有成功处理了，系统才会给你一封确认邮件说是订单成功。我相信有很多朋友都收到认单不成功的邮件。这就是说，数据一致性在并发下是一个瓶颈

异步、throttle（节流阀） 和批量处理都需要对并发请求数做队列处理。

**异步**在业务上一般来说就是收集请求，然后延时处理。在技术上就是可以把各个处理程序做成并行的，也就可以水平扩展了。但是异步的技术问题大概有这些，a）被调用方的结果返回，会涉及进程线程间通信的问题。b）如果程序需要回滚，回滚会有点复杂。c）异步通常都会伴随多线程多进程，并发的控制也相对麻烦一些。d）很多异步系统都用消息机制，消息的丢失和乱序也会是比较复杂的问题。

**throttle 技术**其实并不提升性能，这个技术主要是防止系统被超过自己不能处理的流量给搞垮了，这其实是个保护机制。使用throttle技术一般来说是对于一些自己无法控制的系统，比如，和你网站对接的银行系统。Rate limiter

**批量处理**的技术，是把一堆基本相同的请求批量处理。比如，大家同时购买同一个商品，没有必要你买一个我就写一次数据库，完全可以收集到一定数量的请求，一次操作。这个技术可以用作很多方面。比如节省网络带宽，我们都知道网络上的MTU（最大传输单元），以态网是1500字节，光纤可以达到4000多个字节，如果你的一个网络包没有放满这个MTU，那就是在浪费网络带宽，因为网卡的驱动程序只有一块一块地读效率才会高。因此，网络发包时，我们需要收集到足够多的信息后再做网络I/O，这也是一种批量处理的方式。批量处理的敌人是流量低，所以，批量处理的系统一般都会设置上两个阀值，一个是作业量，另一个是timeout，只要有一个条件满足，就会开始提交处理

Ticket Master: <https://www.jiuzhang.com/qa/630/>

一个类似ticket master的网站。 说某个时间段开放某明星演唱会订票，大概会同时有500K QPS 的访问量，一共只有50K张票。订票的过程是用户打开订票网页（不用考虑认证等问题），填一个text box说要订几张票，然后click一个button就打开一个page，那个page会不停的spin直到系统能够预留那几张票，如果预留成功，用户会有几分钟时间填写用户信息已经完成支付，如果到期未支付，这些票就自动被系统收回了。每张票都是一样的，没有位置信息什么的。

这个题的关键首先是给出一个可行解（无论任何系统设计题的关键都是如此），一个核心要实现的功能是，票的预留与回收。在设计可行解的时候，可以先将500k qps抛之脑后。假设现在只有10个用户来买票。优化的事情，放到Evolve的那一步。

按照我们的SNAKE分析法来：

Scenario 设计些啥:

用户提交订票请求

客户端等待订票

预留票，用户完成支付

票过期，回收票

限制一场演唱会的票数

Needs 设计得多牛？

500k QPS，面试官已经给出

响应时间——是在用户点击的一瞬间就要完成预定么？不是，可以让用户等个几分钟。也就是说，500K的请求，可以在若干分钟内完成就好了。因此所谓的 500k QPS，并不是Average QPS，只是说峰值是 500k QPS。而你要做的事情并不是在1秒之内完成500k的预定，而是把确认你收到了购票申请就好了

Application 应用与服务 (Reservation and Ticket are different things)

ReservationService —— 用户提交一个预定请求，查询自己的预定状态

TicketService —— 系统帮一个预定完成预定，生成具体的票

Kilobyte 数据如何存储与访问

ReservationService —— 用户提交了一个订票申请之后，把一条预定的数据写到数据库里。所以需要一个Reservation的table。大概包含的columns有：

id(primary\_key)

created\_at(timestamp)

concert\_id(foreign key)

user\_id(foreign key)

tickets\_count(int)

status(int)

简单的说就是谁在什么时刻预定了哪个演唱会，预定了几张，当前预定状态是什么（等待，成功，失败）。

2. TicketService —— 系统从数据库中按照顺序选出预定，完成预定，预定成功的，生成对应的Ticket。表结构如下：

id (primary key)

created\_at (timestamp)

user\_id (fk)

concert\_id (fk)

reservation\_id (fk)

status (int) // 是否退票之类的

另外，我们当然还需要一个Concert的table，主要记录总共有多少票：

id (primary key)

title (string)

description (text)

start\_at (timestamp)

tickets\_amount (int)

remain\_tickets\_amount (int)

...

总结一下具体的一个Work Solution 的流程如下：

1. 用户提交一个预定，ReservationService 收到预定，存在数据库里，status=pending

2. 用户提交预定之后，跳转到一个等待订票结果的界面，该界面每隔5-10秒钟像服务器发送一个请求查询当前的预定状态

3. TicketService是一个单独执行的程序，你可以认为是一个死循环，不断检查数据库里是否有pending状态的票，取出一批票，比如1k张，然后顺利处理，创建对应的Tickets，修改对应的Reservation的status。

Evolve

分析一下上述的每个操作在500k qps的情况下会发生什么，以及该如何解决。

1. 用户提交一个预定，ReservationService 收到预定，存在数据库里，status=pending

也就是说，在一秒钟之内，我们要同时处理500k的预定请求，首先web server一台肯定搞不定，需要增加到大概500台，每台web server一秒钟同时处理1k的请求还是可以的。数据库如果只有一台的话，也很难承受这样大的请求。并且SQL和NoSQL这种数据库处理这个问题也会非常吃力。可以选用Redis这种既是内存级访问速度，又可以做持久化的key-value数据库。并且Redis自带一个队列的功能，非常适合我们订票的这个模型。Redis的存取效率大概是每秒钟几十k，那么也就是我们要大概20台Redis应该就可以了。我们可以按照 user\_id 作为 shard key，分配到各个redis上。

2. 用户提交预定之后，跳转到一个等待订票结果的界面，该界面每隔5-10秒钟像服务器发送一个请求查询当前的预定状态

使用了redis的队列之后，如何查询一个预定信息是否在队列里呢？方法是reservation的基本信息除了放到队列里，还需要同时继续存一份在redis里。队列里可以只放reservation\_id。此时reservation\_id可以用user\_id+concert\_id+timestamp来表示。

3. TicketService是一个单独执行的程序，你可以认为是一个死循环，不断检查数据库里是否有pending状态的票，取出一批票，比如1k张，然后顺利处理，创建对应的Tickets，修改对应的Reservation的status。

为每个Redis的数据库后面添加一个TicketService的程序（在某台机器上跑着），每个TicketService负责一个Redis数据库。该程序每次从Redis的队列中读出最多1k的数据，然后计算一下有需要多少张票，比如2k，然后访问Concert的数据库。问Concert要2k的票，如果还剩有那么多，那么就remain\_tickets\_amount - 2k，如果不够的话，就返回还有多少张票，并把remain\_tickets\_amount 清零。这个过程要对数据库进行加锁，可以用数据库自己带的锁，也可以用zookeeper之类的分布式锁。因为现在是1k为一组进行处理，所以这个过程不会很慢，存Concert的数据库也不需要很多，一台就够了。因为我们只有20个跑着TicketService的机器（对应20台Redis），也就是个20 qps的峰值，数据库处理起来绰绰有余。

假如得到了2k张票的额度之后，就顺序处理这1k个reservation，然后对每个reservation生成对应的tickets，并在redis中标记reservation的状态，这里的话，tickets的table大概就会产生2k条的insert，所以tickets的数据库需要大概能够承受 20 x 1k = 20k 的并发写。这个的话，大概 20 台 SQL数据库也就搞定了。

从头理一下

开放订票，500k的请求从世界各地涌来

通过 Load Balancer 纷发给500台 Web Server 。每台Web Server大概一秒钟处理1k的请求

Web Server 将1k的请求，按照 user\_id 进行 shard，丢给对应的 redis 服务器里的队列，并把 Reservation 信息也丢给 Redis存储。

此时，20台 Redis，每台 Redis 约收到 25k 的 排队订票记录

每台 Redis 背后对应一个 TicketService 的程序，不断的查看 Redis 里的队列是否有订票记录，如果有的话，一次拿出1k个订票记录进行处理，问Concert 要额度，然后把1k的reservation对应的创建出2k左右的tickets出来（假如一个reservation有2张票平均）。假如这个部分的处理能力是1k/s的话，那么这个过程完成需要25秒。也就是说，对于用户来说，最慢大概25秒之后，就知道自己有没有订上票了，平均等待时间应该低于10秒，因为当concert的票卖完了的时候，就无需生成1-2k条新的tickets，那么这个时候速度会快很多。

存储tickets的数据库需要多台，因为需要处理的请求大概是20k的qps，所以大概20台左右的Ticket数据库。

超时的票回收

增加一个RecycleService。这个RecycleService 不断访问 Tickets 的数据库，看看有没有超时的票，如果超时了，那么就回收，并且去Concert的数据库里把remain\_tickets\_amount 增加。

总结如何攻破 500k QPS的核心点

核心点就是，500k QPS 我只要做到收，不需要做到处理，那么500台web服务器+20台Redis就可以了。

处理的的时候，分成1k一组进行处理，让用户多等个几秒钟，问题不大。用户等10秒钟的话，我们需要的服务器数目就降低10-20倍，这是个tradeoff，需要好好权衡的。

一些可能的疑惑和可以继续进化的地方

问：500台Web服务器很多，而且除了订票的那几秒种，大部分的时候都是闲置浪费的，怎么办？

答：用AWS的弹性计算服务，为每场演唱会的火爆指数进行评估，然后预先开好机器，用完之后就可以销毁掉。

问：为什么不直接用Redis也来存储所有的数据信息？

答：因为是针对通同一个Concert的预定，大家需要访问同一条数据（remain\_tickets\_amount)，shard是不管用的，Redis也承受不住500k QPS 对同一条数据进行读写，并且还要加锁之类的保证一致性。所以这个对 remain\_tickets\_amount 的值进行修改，创建对应的 tickets 的过程，是不能在用户请求的时候，实时完成的，需要延迟进行。

问：redis又用来做队列，又用来做Reservation 表的存储，是否有点乱？

答：是的，所以一个更好的办法是，只把redis当做队列 和 Reservation(只是中间结果，不大重要，真正重要的是Ticket) 信息的Cache来用。当一个Reservation 被处理的时候，再到SQL数据库里生成对应的持久化记录。这样的好处是，Redis 这种结构其实不是很擅长做持久化数据的存储，我们一般都还是拿来当队列和cache用得比较多。

用redis中的List可以实现队列，这样可以用来做消息处理和任务调度的队列lPush(), rPop(), rPoplPush(list1, list2)

在浏览器地址栏输入URL，按下回车后究竟发生了什么？<http://blog.csdn.net/bruce_6/article/details/39499439>

导航的第一步是通过访问的域名找出其IP地址。DNS查找过程如下：

1.浏览器缓存 – 浏览器会缓存DNS记录一段时间。 有趣的是，操作系统没有告诉浏览器储存DNS记录的时间，这样不同浏览器会储存个自固定的一个时间（2分钟到30分钟不等）。

2.系统缓存 – 如果在浏览器缓存里没有找到需要的记录，浏览器会做一个系统调用（windows里是gethostbyname）这样便可获得系统缓存中的记录。

3.路由器缓存 – 接着，前面的查询请求发向路由器，它一般会有自己的DNS缓存。

4.ISP DNS 缓存 – 接下来要check的就是ISP缓存DNS的服务器。在这一般都能找到相应的缓存记录。

5.递归搜索 – 你的ISP的DNS服务器从跟域名服务器开始进行递归搜索，从.com顶级域名服务器到Facebook的域名服务器。一般DNS服务器的缓存中会有.com域名服务器中的域名，所以到顶级服务器的匹配过程不是那么必要了

静态文件(css, js, photo)会允许浏览器对其进行缓存。有的文件可能会不需要与服务器通讯，而从缓存中直接读取。服务器的响应中包含了静态文件保存的期限信息，所以浏览器知道要把它们缓存多长时间。还有，每个响应都可能包含像版本号一样工作的ETag头.所以，这些文件会在全球很多CDN的数据中心中留下备份.

Http long polling <https://www.pubnub.com/blog/2014-12-01-http-long-polling/>

Web applications were originally developed around a client/server model, where the Web client is always the initiator of transactions, requesting data from the server. Thus, there was no mechanism for the server to independently send, or push, data to the client without the client first making a request.To overcome this deficiency, Web app developers can implement a technique called HTTP long polling, where the client polls the server requesting new information.The server holds the request open until new data is available. Once available, the server responds and sends the new information. When the client receives the new information, it immediately sends another request, and the operation is repeated. This effectively emulates a server push feature.

There are a couple of things to consider when using HTTP long polling to build realtime interactivity in your application, both in terms of developing and operations/scaling.

As usage grows, how will you orchestrate your realtime backend?

When mobile devices rapidly switch between WiFi and cellular networks or lose connections, and the IP address changes, does long polling automatically re-establish connections?

With long polling, can you manage the message queue and catch up missed messages?

Does long polling provide load balancing or failover support across multiple servers?

Although Long Polling works, it is very expensive in terms of CPU, memory and bandwidth. The other approach is WebSocket.

Comet techniques come in two flavors: streaming and long polling. In a Web application using streaming Comet, the browser opens a single persistent connection to the server for all Comet events and handles them incrementally on the browser side. Specific examples of streaming Comet techniques include the Hidden IFrame and the XMLHttpRequest Server Push. In long polling, the client makes an Ajax request to the server, which is kept open until the server has new data to send to the browser. Upon receiving the server response, the browser initiates a new long polling request in order to obtain the next data set. Long polling can be achieved using either Ajax or script tag techniques.

A single instance of Caplin liberator can support up to 100,000 clients each receiving 1 message per second with an average latency of less than 7ms. How does this to compare to HTML5 websockets on any webserver?

We could split up the methods we are talking about into three camps..

1.Comet HTTP polling - including long polling

2.Comet HTTP streaming - server to client messages use a single persistent socket with no HTTP header overhead after initial setup

3.Comet WebSocket - single bidirectional socket

I see them all as Comet, since Comet is just a paradigm, but since WebSocket came along some people want to treat it like it is different or replaces Comet - but it is just another technique - and unless you are happy only supporting the latest browsers then you can't just rely on WebSocket.

As far as performance is concerned, most benchmarks concentrate on server to client messages - numbers of users, numbers of messages per second, and the latency of those messages. For this scenario there is no fundamental difference between HTTP Streaming and WebSocket - both are writing messages down an open socket with little or no header or overhead.

Long polling can give good latency if the frequency of messages is low. However, if you have two messages (server to client) in quick succession then the second one will not arrive at the client until a new request is made after the first message is received.

I think someone touched on HTTP KeepAlive. This can obviously improve Long polling - you still have the overhead of the roundtrip and headers, but not always the socket creation.

GFS: <http://blog.csdn.net/koder2009/article/details/3964878>

GFS简单来说，有两个部分，一个是Master，一个是ChunkServer。Master可以理解为一个总控主机，而ChunkServer，则是一系列的主机，存储实际上的数据，并且接收Master的监控。所以，这是一个（1：many）的关系

在GFS系统中，一个基本理念是：尽量处理大块数据，并且尽量优化大块数据的处理，而不去优化小块数据的处理。所以，一个基本的处理单位，是64MB，这就是文件块（Chunk）的概念。一个ChunkServer上面会有很多的Chunk（数据块），这些数据块都是64MB大小，由ChunkServer负责读写。

从职能上来说，Master维护的信息，就好像是普通PC机的目录表，它记录了所有的chunk块的位置（即对应的ChunkServer的地址，如IP：PORT），这样做的基础，是每一个chunk块由一个全局性的GUID，这同样是由Master在创建这个文件块的时候分配的。所以，当用户需要进行操作的时候，会先去Master主机获取对应的目录信息，得到对应的ChunkServer，然后，真正向ChunkServer读写数据。

那么，master是如何管理这些chunkServer的呢？google的做法非常简单直接。master不会存储chunkserver的信息，而是在每一次启动的时候，向cluster中的所有chunkserver发送指令，由所有的chunkserver自动汇报自己的信息，包括自己有的数据块对应的GUID等，运行状态等，这些数据将会存放到master的内存中（不是硬盘），而master在启动之后，内存中不久就会有所有chunkserver的数据，之后，它会定期去发送消息到chunkserver，以确保这个chunkserver还处于工作状态，通过这种方式，能够及时发现down掉的机器，并且执行容错处理。而当加入新的机器上，只需到master注册一下，新的机器立刻可以加入cluster开始工作，这就是其可扩展性的伟大之处。

那么GFS是如何容错的呢？实际上，每一个chunk数据块有3份拷贝，一份down掉，则GFS会立刻通过另外两份快速进行恢复。通过这种方式，可以把出错的几率降低到一个几乎可以忽略的程度。

另外，值得一提的是，GFS的客户端，它和master通信的数据，只是传递控制信息，比如获取目录即chunkserver地址，但是数据传输不会经过master，而是直接和chunkserver通信，这种方式，可以大大降低master的负载，并且容易做到全局负载均衡

BigTable: <http://blog.csdn.net/koder2009/article/details/3985329>

Bigtable的目的，是要提供一个能够快速处理大规模结构化数据的能力。它是基于多维稀疏map的一个典型应用。从逻辑上，它的数据模型非常简单：<row, column, timestamp>--<value>这样的数据项。即行，列，时间戳同时作为键值，而任意字符串作为内容。在进行实际的存储管理的时候，除了把数据一维化<Key-Value>，还把数据按照行来进行分片。一般来说，一个bigtable的行是按照顺序存放的，而在每隔一定量的数据行，就会划分成为一个tablet的小分片，因此，一个大的bigtable通常是有很多的tablet组成的，这些tablet将会分散到GFS管理的集群上，每一个机器上有若干个tablet，保持每一个机器的容量和负载不至于过大。

一个tablet的存放也是非常讲究的，为了实现高效的存取，bigtable使用了一种简单的结构SStable，来完成实际的存储。这种sstable，实际就是一系列的key-value对，这些key-value是连续存储成块（符合GFS特性），并且这些key-value对会按照64k的方式来分成一个个block，所以，一个sstable，通常有若干个block，并且一个sstable会维护一个index数据信息，这些信息就是每一个block的起始键值以及对应在这个sstable中的偏移量。这样，就可以快速定位，快速读取数据。

SSTable 是在内存中排序好写入的。  
关于在SSTable中查找和读的问题，因为SSTable有很多个sharding（block），找key对应的数据的时候，先会通过hash函数计算key会落在哪个sharding当中，单独一个sharding读取的时候，打开之后是全部加载进入内存的。

SSTable 是 immutable 的. key（row + col）有可能在多个 SSTable 里面, 因为所有的 modification (包括 deletion) 都会是 append.

Key = row\_key + col\_key + timestamp. 可以用Partial index搜索，例如（row\_key+col\_key）

MapReduce: <http://blog.csdn.net/koder2009/article/details/3991938>

mapreduce主要是两个过程，一个是map过程，负责把一个庞大的任务，细分成为一个小任务，然后分配到不同的服务器上运行。而reduce过程，则是负责把已经细分的任务的计算结果，重新合并成为想要的完整结果

1. 客户端提交任务，并且指定map和reduce的具体策略。

2. master（mapreduce的中控系统）接收到任务，按照map策略，把任务划分为细的任务，并且，相应的数据也进行了正确的划分。

3. master根据任务数量，从拥有空闲计算资源的worker（计算机）中，选择一部分进行细分了的任务的具体计算，同时选定一部分作为reducer。

4. 负责完成细分任务的机器从master处获取数据，开始任务的计算过程，并且将计算结果暂存在本地的磁盘上。

5. 负责完成细分任务的机器完成之后，同时master则按照一定的策略，通知reducer去收集那些暂存的结果进行处理。

6. reducers从各个拥有暂存结果的机器上，读取中间结果数据，并且按照指定的reduce策略，计算获得最终的结果，并且存放到指定的位置，供客户端读取。通知master任务的进度已经完成。Reducer的输入是在硬盘（不是内存）里面， load 一小部分，写完了后，继续load一小部分

7. master通知客户端计算结束，客户端读走结果数据。

Large scale key-value lookup service design: <https://www.jiuzhang.com/qa/627/>

设计一个只读的lookup service. 后台的数据是10 billion个key-value pair, 服务形式是接受用户输入的key，返回对应的value。已知每个key的size是0.1kB，每个value的size是1kB。要求系统qps >= 5000，latency < 200ms.

server性能参数需要自己问，我当时只问了这些，可能有需要的但是没有问到的:

commodity server

8X CPU cores on each server

32G memory

6T disk

使用任意数量的server，设计这个service

Google 有一个基础的数据结构 SSTable(Sorted String Table), 是一个简单的抽象，用来高效地存储大量的键-值对数据，同时做了优化来实现顺序读/写操作的高吞吐量。SSTable 就是用来支持大量的读操作。这个设计题就是让你设计SSTableService

Given 10 billion key-value pair

=> total key size ~ 10 billion \* 0.1kB = 1T

=> total value size ~ 10 billion \* 1kB = 10T

Since it's read only, so SSTable is suitable in this case rather than NoSQL.

with 6T disk , a server with two disks will be enough.

For every request, 1 value, which is 1kB needs to be returned.

According to <https://fusiontables.google.com/DataSource?snapid=S523155yioc>

total time for reading one value will be 10ms(disk seek) + 1kB/1MB \* 30ms(reading 1kB sequentially from disk) = 10ms.

QPS on 1 server will be 1s/10ms \* 2 disk = 200

required QPS support is 5000. So we need 5000/200 = 25 servers.

And for latency, there are several things need to be considered: finding the key, read the value, return the value.

Using binary search, we need log(n) times to find the key. For each time, the disk latency is 1 seek plus 1 read, reading key is really small, so can be ignored. So total time for find the key is log(10billion) \* 10ms = 100ms.

Reading a key will take another disk seek , 10ms.

1 round trip in the same data center is 0.5ms.

Assume network bandwidth is 1Gbps, sending 1kB will take very short time, so it's ignored.

so total latency is 100 + 10 + 0.5 = 110.5ms.

News Feed: <https://www.quora.com/Software-Engineering-Best-Practices/What-are-the-best-practices-for-building-something-like-a-News-Feed>

Users in most social networking sites are described in terms of a **social graph**. The relationships between users are represented by adjacency lists. If Jack and Jill are friends, they are said to be adjacent. This is known as an "**edge**" in the graph.

Determining Importance

You'll likely want to rank edges by importance rather than simply the most recent updates, meaning that you need to calculate some sort of **score**. Facebook's EdgeRank was described by the formula ∑e = ue\*we\*de, wherein ∑e is the sum of the edge's rank, ue is the **affinity score** with the user who created the edge, we is the **weight for the content type**, and de is a **time decay factor**.

Calculating a friend's affinity score can be done something like this: ∑i = li\*ni\*wi, wherein ∑i is the sum of the interactions with that friend, li is the **time** since your last interaction (this would need to be weighted so that 1 day > 30 days), ni is the number/**count** of interacts, and wi is the **weight** of those interactions. This method allows you to rank friends in a separate database and then perhaps only show ten updates from the ten closest friends, which isn't a bad idea considering few of us are likely to have more close friends than this.

You want to minimize the number of disk seeks that need to happen when loading your home page. The number of seeks could be 0 or 1 but definitely not O(num of friends). You also can't store all the data on one machine if you're concerned about scaling, so you've got a couple of options...

If you're willing to tolerate one disk seek, or if your graph has low fan-out (small number of people following any given person), you can de-normalize the data such that the metadata about every piece of activity is propagated to each of the followers of that activity at the time the action occurs. You might think of this as a "**push**" model. You'd still probably only store one copy of the actual activity data, but you'd push pointers to it (along with whatever other metadata is needed if you're supporting any ranking/filtering) to all the subscribers at the time it is created. Generally the first thing to break in this model will be the process of propagating the activity to all the subscribers, particularly if you have users that have large numbers of followers (celebrities). This can also be complicated in that you may need to write code that properly updates all of the subscribers whenever the important metadata about the content is updated（支持用户edit自己的activity? No, but user can delete,and you may want to also add code to update things when someone changes their list of subscriptions(我不关注他了，把他的Activity从我的user feed删除).

The alternative is to keep all the recent activity data in **memory** and not propagate the updates to the subscribers at write time, instead fanning out at the time of loading the home page. This way you avoid all disk seeks. It's also nice in that your fan out size is limited to the number of people a user follows rather than the number of people who follow a user (most people don't have enough time on their hands to follow millions of people, so you don't have the inverse of the celebrity problem). It's also easier to keep things up-to-date, since you don't have to worry about propagating updates to all of the subscribers.The downside of this approach is that the failure scenario is more catastrophic - instead of just delaying updates, you may potentially fail to generate a user's feed. Having some kind of fallback mechanism that approximates the feed (eg by querying only a subset of your friends) is handy to avoid having to show an error page.

A user timeline is all the tweets a particular user has sent.

A home timeline is a temporary merge of all the user timelines of the people are you are following.

为什么Facebook, Twitter都从Push转成了Pull:

1. Push 无法很好的解决僵尸粉问题.Push 花了很多时间去 deliver 帖子给很多不活跃用户，让整个系统花了很多算力在做无用功上，及其浪费。
2. Push 在明星的帖子上 Deliver 太慢，用户看帖有延迟
3. Push 无法很好的解决 News Feed 重新排序的问题.Twitter 和 Facebook 的 News Feed (or Timeline) 越来越复杂，不仅仅是简单的按照时间排序了，会整合更多的资源，比如新闻，比如热帖，比如广告。这些内容都需要（且只能）通过 Pull 得到。
4. pull可以加入cache timeline + newsfeed 的方法。这样每个用户一次读最新的newfeed操作就等于用 cache timeline 中的上一次访问时间戳到当前访问时间戳中新增加的tweets来更新cache 中的newsfeed。这种方法似乎完全取代了push model读的时候快的好处. Pull 就是用内存来做了 NewsFeedTable。所以相比之下，就更费内存一些

首先，数据库请求是分级的。你只要最新5-10分钟的数据的话，没多少，速度很快的。

其次，你要最新5-10分钟的数据也是去cache里拿，cache拿不到才去DB。

总结：这里Cache有两不同类型的数据，一个是每个用户的 timeline，一个是每个用户的Newsfeed。你只理解了 Newsfeed要cache，没有理解timeline要cache。

所以每次发帖的时候会：更新自己的timeline cache

每次登陆会：pull following的 timeline, merge, 更新自己的 newsfeed cache

How to Cache timeline:

key是user\_id加一个prefix表示这是存的timeline（比如 "timeline::123") ，value是 list of tweet ids。不存具体的每一条tweet的具体内容，只存id。然后还会有一个key为 "tweet::345"，value为具体的一条tweet内容的存在cache里。

user post的时候： invalidate cache -> update db 。这个时候并不update cache。只有当有人需要他的 timeline的时候，去读的时候，才 read from db -> update cache。

归并N个用户的在某个时间戳之后的所有Tweets.这个timestamp可以存在client，也就是手机上，或者web端。这个请求是客户端发起的，客户端会知道自己现在的数据最新的一条是什么时候的。这个时候的key就可以是 "newsfeed::123"，所以这就是为什么要加prefix，否则你怎么区别是 timeline 里的东西还是 newsfeed的东西呢。

"每次read cache + 归并完之后update这个cache吗？"-> 对的。

**NewsFeed如何实现pagination:**

首先不会有帖子的timestamp一样，timestamp的精度很高的（微秒级别）

翻页并不是简单的1-100，101-200这样去翻页。因为当你在翻页的时候，你的news feed可能已经添加了新的内容，这个时候你再去检索最新的101-200可能和你的1-100就有重叠了。

通常的做法是，拿第101个帖子的timestamp作为下一页的起始位置，也就是说，当用户在看到第一页的前100个帖子的时候，他还有第101个帖子的timestamp信息（隐藏在你看不到的地方），然后你请求下一页的时候，会带上这个timestamp的信息，server端会去数据库里请求 >= timestamp 的前101个帖子，然后也同样把第101个帖子作为下一页的timestamp。这个方法比直接用第100个帖子的timestamp好的地方是，你如果读不到第101个帖子，说明没有下一页了，如果你刚才只有100个帖子的话，用第100个帖子的timestamp的坏处是，你会有一次空翻。？？？

**News Feed，所有用户共用一个TimeLine Table和NewsFeed Table(Push). 可能会根据UserID sharding?**

Pull 模型是没有 News Feed Table的，Pull 只有 Timeline table。

我的 NewsFeed Table 存的是，谁发了什么，谁可以看到。包含三个部分，owner\_user\_id（对方，这个newsfeed 是发给谁的），user\_id 这个内容是谁发的（也可以不存这个，因为 可以根据 tweet\_id 去查是谁发的），然后 tweet\_id，然后一些其他的内容。

而 Timeline Table 存的是，谁发了什么。也就是主要包含2个部分，user\_id和发的内容。也就是说，如果你不做任何优化的话，实际上 Timeline Table 就是 Tweet Table。因为你可以 select \* from tweet\_table where user\_id=某人，这就是某人的 timeline 了。

当你发了一个帖子之后，如果是 pull 模型，那么只需要在 Tweet Table 里增加一个你发的帖子的记录就好了。其他什么也不用做，当你的好友需要看你的帖子的时候，主动去找你发过的最近100条什么的。

而 Push Model 下，你发了一个tweet之后，系统需要主动的 deliver你的这个帖子去到 newsfeed table 里去。比如你有3个好友A,B,C。那么系统需要往 news feed table 里存入 [A+你的帖子], [B+你的帖子], [C+你的帖子] 三条数据。

**Edit功能.除了 FACEBOOK 之外，其他大型社交网站都没提供 EDIT 功能.**

Edit 的难点在于，企业为了加速用户的读取效率，在帖子被分享（或者分享之后再分享），被 deliver 到 news feed 的时候，大量使用了我们课上讲到的 "Denormalize" 这个技术。也就是说，帖子会被复制一份，然后写在各种 DB 和 Cache 里。那么此时，如果你需要进行 In-place 的修改，就需要去把所有的这些你可能无法追踪到了的帖子在 DB 和 Cache 里的所有“副本” 进行修改。但是这个就很难做到了。一则是过程缓慢，第二则是如果你要追踪这些帖子的话，那么在一开始放进 DB 和 Cache 里的时候，可能就需要去建立相应的 反向查询的 index 之类的，但是这样带来的耗费就很大。

从另外一个角度，目前的用户都可以使用 delete + repost 的方式进行 “edit“ 所以从某种意义上已经满足了用户 edit 的需求，再花大力气去做一个 edit 的功能就未必划算了。Facebook 造了很多轮子，能解决很多很难的问题，比如说支持放进 DB 和 Cache 里的时候，去建立相应的 反向查询的 index, 来支持edit功能的实现。消耗可能很大，但是facebook有足够的resources去支持这个巨大的消耗。

WebSocket: <https://www.zhihu.com/question/20215561>

ajax轮询 需要服务器有很快的处理速度和资源。（速度）

long polling 需要有很高的并发，也就是说同时接待客户的能力。（场地大小）以上两种传统方式又称为Comet

WebSocket 我们有一个非常快速的接线员（Nginx），他负责把问题转交给相应的客服（Handler）。本身接线员基本上速度是足够的，但是每次都卡在客服（Handler）了，老有客服处理速度太慢，导致客服不够。Websocket就解决了这样一个难题，建立后，可以直接跟接线员建立持久连接，有信息的时候客服想办法通知接线员，然后接线员在统一转交给客户。这样就可以解决客服处理速度过慢的问题了。同时，在传统的方式上，要不断的建立，关闭HTTP协议，由于HTTP是stateless的，每次都要重新传输identity info（鉴别信息），来告诉服务端你是谁。虽然接线员很快速，但是每次都要听这么一堆，效率也会有所下降的，同时还得不断把这些信息转交给客服，不但浪费客服的处理时间，而且还会在网路传输中消耗过多的流量/时间。但是Websocket只需要一次HTTP握手，所以说整个通讯过程是建立在一次连接/状态中，也就避免了HTTP的非状态性，服务端会一直知道你的信息，直到你关闭请求，这样就解决了接线员要反复解析HTTP协议，还要查看identity info的信息的问题。同时由客户主动询问，转换为服务器（推送）有信息的时候就发送（当然客户端还是等主动发送信息过来的），没有信息的时候就交给接线员（Nginx），不需要占用本身速度就慢的客服（Handler）了

WebSocket is designed to be implemented in web browsers and web servers, but it can be used by any client or server application. The WebSocket Protocol is an independent TCP-based protocol. Its only relationship to HTTP is that its handshake is interpreted by HTTP servers as an Upgrade request.The WebSocket protocol enables interaction between a browser and a web server with lower overheads, facilitating real-time data transfer from and to the server. This is made possible by providing a standardized way for the server to send content to the browser without being solicited by the client, and allowing for messages to be passed back and forth while keeping the connection open. In this way, a two-way (bi-directional) ongoing conversation can take place between a browser and the server. The communications are done over TCP port number 80 (or 443 in the case of TLS-encrypted connections), which is of benefit for those environments which block non-web Internet connections using a firewall.

Trie: <https://en.wikipedia.org/wiki/Radix_tree>

A radix trie (also compact prefix tree) is a data structure that represents a space-optimized trie in which each node that is the only child is merged with its parent. The result is that the number of children of every internal node is at least the radix r of the radix tree, where r is a positive integer and a power x of 2, having x ≥ 1. Unlike in regular tries, edges can be labeled with sequences of elements as well as single elements. This makes radix trees much more efficient for small sets (especially if the strings are long) and for sets of strings that share long prefixes.

Auto Complete: <https://stackoverflow.com/questions/2901831/algorithm-for-autocomplete>

You probably want a way to bound the size of the trie:

1.Keep a trie of recent/top N words used globally;

2.For each user, keep a trie of recent/top N words for that user.

You want to prevent lookups whenever possible...

1.Cache lookup results: if the user clicks through on any search results, you can serve those very quickly and then asynchronously fetch the full partial/fuzzy lookup.

2.Precompute lookup results: if the user has typed "appl", they are likely to continue with "apple", "apply".

3.Prefetch data: for instance, a web app can send a smaller set of results to the browser, small enough to make brute-force searching in JS viable.

3 different auto-complete implementations <http://sujitpal.blogspot.com/2007/02/three-autocomplete-implementations.html>

\* In-Memory Trie

\* In-Memory Relational Database, Like %abc%

\* Java TreeSet: treeSet.tailSet(prefix) Returns a view of the portion of this set whose elements are greater than or equal to fromElement.

When looking up keys, the trie is marginally faster than the TreeSet implementation. Both the Trie and the TreeSet are a good bit faster than the relational database solution.The setup cost of the Set is lower than the Trie or DB solution. You'd have to decide whether you'd be constructing new "wordsets" frequently or whether lookup speed is the higher priority.

在Design google suggestion时,当Trie无法在一个server的memory中放下的话，我们应该sharding. Sharding key是什么？

首先你需要一个log 的service，把用户的搜索信息记录下来，比如：

{"apple": 1, "google": 2, "baidu": 3}

这个时候key是用户搜索的完整单词，value 是搜索次数。

然后用一个离线脚本，将这个结果整理为Trie。这个时候我们称之为Trie'吧。在Trie'中，每个TrieNode上也只存储到此节点为一整个单词的被搜索的次数。

接着你需要再用一个离线脚本去统计，每个Prefix 下面的Top 10的结果是什么。完成这个操作之后的Trie 我们称之为Trie'' ，也就是此时，TrieNode 上不仅仅存储了该Prefix 被搜索的次数，还存储了以该Prefix 开头的Top 10 的单词。举个例子：

{

"a": ["ab", "ac", "ad" ...]

-- "ab": ["abc", "abe" ..]

-- "ac": ["aca", "acd" ..]

"b": ...

...

}

也就是说，这个时候的Trie''，在给定某个prefix的时候，就能够直接在O(Len(prefix)) 的时间复杂度内找到Top K，而不需要遍历整个Trie 去搜寻。高频的topk 词汇，我们做不到实时，是伪实时。日志每十五分钟收集统计出词频。再走数据配送到index服务，数据热更新。因为有了离线计算过程，所以数据都是pre-sort的，加载到内存时已经排好序

然后此时我们发现内存不够了，要开始sharding了，我们可能会把"a" 和“ab” 分在不同的机器上，但是a的那个小的top 10 的list，也就是["ab", "ac", "ad" .. ] 会跟着a走，也就是虽然ab ac ad 作为key的那个prefix，不和a在同一台机器上，但是以a作为prefix开头的top 10 的结果，已经在Trie'' 中计算好了，所以跟着a走就可以了。直接shard每个节点到不同的机器上面

假设根据hash, hash(a)=1 hash(ab)=2 hash(abc)=3, 也就是如果query string is “a”, go to server #1, if query string is “ab”, go to server #2. a虽然在哪儿都有，但是a在2和3上是只是中间节点，只有在1上才是有单词的节点.

Update可能需要在最原始的Trie上进行更新，之后通过offline job更新Trie''

Typeahead in Facebook: <https://www.facebook.com/notes/facebook-engineering/the-life-of-a-typeahead-query/389105248919/>

1. Bootstrapping Connections

As soon as the user focuses in the text box, we send off a request to retrieve all of the user's friends, pages, groups, applications, and upcoming events. We load these results into the **browser's cache**, so that the user can find these results immediately without sending another request. The old typeahead did this, but stopped here.

2. AJAX Request

If there are not enough results in the current browser cache, the browser sends an AJAX request containing the current query string and a list of results that are already being displayed due to browser cache. Our load balancer routes the request to an appropriate web machine.

3. Aggregator Service

The php AJAX endpoint is a thin wrapper around a Thrift service for handling typeahead queries. This service, called the "aggregator", is at the root of a tree of separate search services. Aggregators themselves are stateless and have no index of their own. They are instead responsible for delegating queries to multiple lower-level services in parallel and integrating their results.

4. Leaf Services

In parallel, several different backend search services scour their indices for results that prefix-match the contents of the search box. Each leaf service is designed to retrieve and rank results on only a few specific features. The architecture of the system allows us to add new services, too, as we discover important sources of results and features.

The **global service** maintains an index of all the pages and applications on the site. Unlike most of what we do at Facebook, these results are global because they contain no personalization. The signals that this service records are identical for all user on the site; for example, we might rank applications by the number of users who have interacted with them in the last month, and pages by the structure of the graph surrounding them. Since this service's results are independent of the querying user, we can save latency by storing recent results in a memcached-based query cache.

The **graph service** returns results by searching the user's neighborhood of the graph. For some types of queries, a user and her friend's graph connections are a powerful signal of expressed preferences, and thus relevance. Graphs are notoriously difficult mathematical objects for computers to come to terms with: they are large, and are a minefield of computationally hard problems that appear simple. Our graph contains 400 million active users, and many billions of connections, both among users and from users to objects of other types: pages, applications, events, open graph nodes, etc.

5. Merging Results

The aggregator merges the results and features returned from each leaf service and ranks the results according to our model. The top results are returned to the web tier.

1. Fetching Data and Validating Results

The results returned by the aggregator are simply a list of ids. The web tier needs to fetch all the data from memcache/MySQL to render the results and display information like the name, profile picture, link, shared networks, mutual friends, etc. The web tier also needs to do **privacy checking** here to make sure that the searcher is allowed to see each result.

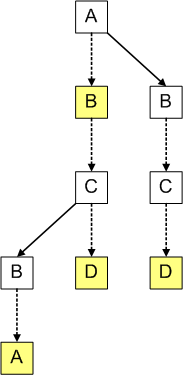
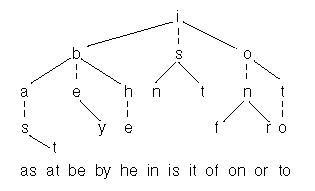
7. Displaying the Results

The results with all the relevant data are sent back to the browser to be displayed in the typeahead. These results are also added to the browser cache along with the bootstrapped connections so that similar subsequent queries don't need to hit the backend again.

Ternary search tree: <http://www.drdobbs.com/database/ternary-search-trees/184410528>

Ternary search trees combine attributes of binary search trees and digital search tries. Like tries, they proceed character by character. Like binary search trees, they are space efficient, though each node has three children, rather than two. A search compares the current character in the search string with the character at the node. If the search character is less, the search goes to the left child; if the search character is greater, the search goes to the right child. When the search character is equal, though, the search goes to the middle child, and proceeds to the next character in the search string.

Figure 3 is a balanced ternary search tree for the same set of 12 words. The low and high pointers are shown as solid lines, while equal pointers are shown as dashed lines. Each input word is shown beneath its terminal node. A search for the word "is" starts at the root, proceeds down the equal child to the node with value "s," and stops there after two comparisons. A search for "ax" makes three comparisons to the first letter ("a") and two comparisons to the second letter ("x") before reporting that the word is not in the tree. 上边有竖线才代表相等，有word

AB, ABCD, ABBA, BCD

<http://igoro.com/archive/efficient-auto-complete-with-a-ternary-search-tree/>

Implementing auto complete using a trie is easy. We simply trace pointers to get to a node that represents the string the user entered. By exploring the trie from that node down, we can enumerate all strings that complete user’s input. If we needed to support all 26 English letters, each node would have to store 26 pointers. And, if we need to support international characters, punctuation, or distinguish between lowercase and uppercase characters, the memory usage grows becomes untenable.

Our problem has to do with the memory taken up by all the null pointers stored in the node arrays. We could consider using a different data structure in each node, such as a hash map. However, managing thousands and thousands of hash maps is generally not a good idea, so let’s take a look at a better solution.

A ternary tree is a data structure that solves the memory problem of tries in a more clever way. To avoid the memory occupied by unnecessary pointers, each trie node is represented as a tree-within-a-tree rather than as an array. Each non-null pointer in the trie node gets its own node in a ternary search tree.

The ternary search tree contains three types of arrows. First, there are arrows that correspond to arrows in the corresponding trie, shown as dashed down-arrows. Traversing a down-arrow corresponds to “matching” the character from which the arrow starts. The left- and right- arrow are traversed when the current character does not match the desired character at the current position. We take the left-arrow if the character we are looking for is alphabetically before the character in the current node, and the right-arrow in the opposite case.

缓存更新的套路 <http://coolshell.cn/articles/17416.html>

Cache Aside Pattern这是最常用最常用的pattern了。其具体逻辑如下：

失效：应用程序先从cache取数据，没有得到，则从数据库中取数据，成功后，放到缓存中。

命中：应用程序从cache中取数据，取到后返回。

更新：先把数据存到数据库中，成功后，再让缓存失效。

注意，我们的更新是先更新数据库，成功后，让缓存失效。为什么不是写完数据库后更新缓存？主要是怕两个并发的写操作导致脏数据。

Just imagine what if two concurrent updates of the same data element occur? You might have different values of the same data item in DB and in memcached. Which is bad. There is a certain number of ways to avoid or to decrease probability of this. Here is the couple of them:

1. A single transaction coordinator

2. Many transaction coordinators, with an elected master via Paxos or Raft consensus algorithm

3. Deletion of elements from memcached on DB updates

I assume that they chose the way #3 because "a single" means a single point of failure, and Paxos/Raft is not easy to implement plus it sacrifices availability for the benefit of consistency.

一个是查询操作，一个是更新操作的并发，首先，没有了删除cache数据的操作了，而是先更新了数据库中的数据，此时，缓存依然有效，所以，并发的查询操作拿的是没有更新的数据，但是，更新操作马上让缓存的失效了，后续的查询操作再把数据从数据库中拉出来。而不会像文章开头的那个逻辑产生的问题，后续的查询操作一直都在取老的数据。

那么，是不是Cache Aside这个就不会有并发问题了？不是的，比如，一个是读操作，但是没有命中缓存，然后就到数据库中取数据，此时来了一个写操作，写完数据库后，让缓存失效，然后，之前的那个读操作再把老的数据放进去，所以，会造成脏数据。但，这个case理论上会出现，不过，实际上出现的概率可能非常低，因为这个条件需要发生在读缓存时缓存失效，而且并发着有一个写操作。而实际上数据库的写操作会比读操作慢得多，而且还要锁表，而读操作必需在写操作前进入数据库操作，而又要晚于写操作更新缓存，所有的这些条件都具备的概率基本并不大。

解决或者缓解data race问题， 将ttl设置为1天，这样即使出现不同，也会在一天之后失效。这个叫做 eventually consistent。是常用的办法，cache aside 是无法根本的避免 race condition 的。所以通常都是 eventually consistent。

Read/Write Through Pattern

我们可以看到，在上面的Cache Aside套路中，我们的应用代码需要维护两个数据存储，一个是缓存（Cache），一个是数据库（Repository）。所以，应用程序比较啰嗦。而Read/Write Through套路是把更新数据库（Repository）的操作由缓存自己代理了，所以，对于应用层来说，就简单很多了。可以理解为，应用认为后端就是一个单一的存储，而存储自己维护自己的Cache。

Read Through 套路就是在查询操作中更新缓存，也就是说，当缓存失效的时候（过期或LRU换出），Cache Aside是由调用方负责把数据加载入缓存，而Read Through则用缓存服务自己来加载，从而对应用方是透明的。

Write Through 套路和Read Through相仿，不过是在更新数据时发生。当有数据更新的时候，如果没有命中缓存，直接更新数据库，然后返回。如果命中了缓存，则更新缓存，然后再由Cache自己更新数据库（这是一个同步操作）

Write Behind Caching Pattern 又叫 Write Back

Write Back套路，一句说就是，在更新数据的时候，只更新缓存，不更新数据库，而我们的缓存会异步地批量更新数据库。这个设计的好处就是让数据的I/O操作飞快无比（因为直接操作内存嘛 ），因为异步，write back还可以合并对同一个数据的多次操作，所以性能的提高是相当可观的。

但是，其带来的问题是，数据不是强一致性的，而且可能会丢失（我们知道Unix/Linux非正常关机会导致数据丢失，就是因为这个事）。在软件设计上，我们基本上不可能做出一个没有缺陷的设计，就像算法设计中的时间换空间，空间换时间一个道理，有时候，强一致性和高性能，高可用和高性性是有冲突的。软件设计从来都是取舍Trade-Off。另外，Write Back实现逻辑比较复杂，因为他需要track有哪数据是被更新了的，需要刷到持久层上。操作系统的write back会在仅当这个cache需要失效的时候，才会被真正持久起来，比如，内存不够了，或是进程退出了等情况，这又叫lazy write。

上面，我们没有考虑缓存（Cache）和持久层（Repository）的整体事务的问题。比如，更新Cache成功，更新数据库失败了怎么吗？或是反过来。关于这个事，如果你需要强一致性，你需要使用“两阶段提交协议”——prepare, commit/rollback，比如Java 7 的XAResource，还有MySQL 5.7的 XA Transaction，有些cache也支持XA，比如EhCache。当然，XA这样的强一致性的玩法会导致性能下降。

Hotspot(也就是Thundering Herds吧)问题指的是而对于同一条数据库记录（例如 一条tweet） 的反复多用户操作（like，comment）的问题，对于这种问题可以再深入从cache 方法上解决.Hotspot是一个应用Read Through / write through 的例子.

Distributed Cache system 一般有2 种 design partten：

1. Cache aside

Cache aside 要求Application 来管理cache 内容，保证数据一致性；当读取时，直接读取Cache；如果没有数据或数据invalid，那么就要读datastore，读完以后再用datastore的值写入cache。 当写入时，先写Data store，再invalid cache.(下一次读就强制先读Datastore)。

memcached的使用建议先delete再来set 如果先set memcached，然后db crash，将来memcached和db数值就不一样了 脏了

如果先delete，db crash最多就是memcached没有值 没关系

2. Read Through/Write Through

Application 层就把 cache 当datastore， cache 的update 由cache 自己保证； cache 会通过和data store 的异步操作来auto refresh自己，这样cache自动就是最新的， 而且由于是异步操作，避免了peak time query 过多的问题。

当Lady Gaga 发新推文的时候，很多用户like,comment,造成同一条推文反复修改； 当采用WT 模式的时候，推文直接在cache 里面修改，立即返回用户，然后cache 在DB 不忙的时候再更新DB； 这样就降低了latency ，并且减少了 DB峰值压力。

cache aside：优点可以为保证了数据的一致性。无论发生何种failure，cache和db的数据是一致的。缺点：数据可能在cache中会短暂时间不存在，导致cache miss。当然在某些高请求的情况下，会有thundering herds出现。

cache through：优点是更新数据现在cache中出现，再高并发的情况下更能胜任。但可能造成和db数据的不一致性。但如果更新不是重要变化的话，有点latency没问题。

Cache Aside 装几个软件就可以实现了（Memcached+MySQL)。能提供 Cache Through 支持的目前只有 Redis 并且 Redis是单机不容易扩展

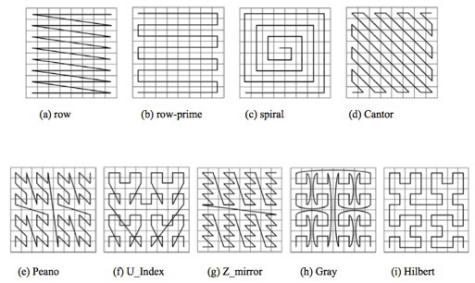
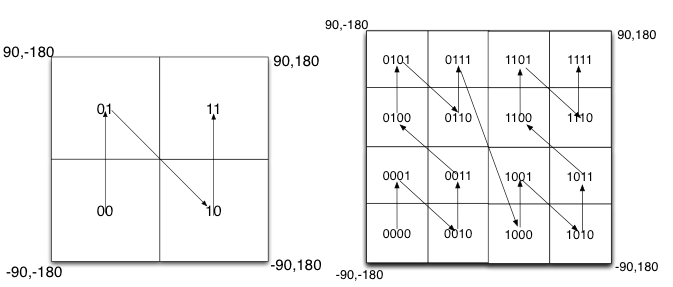
GeoHash: <http://www.cnblogs.com/LBSer/p/3310455.html>

寻找附近餐馆的计算思想很朴素，就是通过过滤的方法来减小参与计算的餐馆数目，从某种角度上讲，需要使用索引技术。一提到索引，大家脑子里马上浮现出B树索引，因为大量的数据库（如MySQL、oracle、PostgreSQL等）都在使用B树。B树索引本质上是对索引字段进行排序，然后通过类似二分查找的方法进行快速查找，即它要求索引的字段是可排序的，一般而言，可排序的是一维字段，比如时间、年龄、薪水等等。但是对于空间上的一个点（二维，包括经度和纬度），如何排序呢？又如何索引呢？如果能通过某种方法将二维的点数据转换成一维的数据，那样不就可以继续使用B树索引了嘛

1）GeoHash将二维的经纬度转换成字符串，比如下图展示了北京9个区域的GeoHash字符串，分别是WX4ER，WX4G2、WX4G3等等，每一个字符串代表了某一矩形区域。也就是说，这个矩形区域内所有的点（经纬度坐标）都共享相同的GeoHash字符串，这样既可以保护隐私（只表示大概区域位置而不是具体的点），又比较容易做缓存，比如左上角这个区域内的用户不断发送位置信息请求餐馆数据，由于这些用户的GeoHash字符串都是WX4ER，所以可以把WX4ER当作key，把该区域的餐馆信息当作value来进行缓存，而如果不使用GeoHash的话，由于区域内的用户传来的经纬度是各不相同的，很难做缓存。

2）字符串越长，表示的范围越精确。5位的编码能表示10平方千米范围的矩形区域，而6位编码能表示更精细的区域（约0.34平方千米）

3）字符串相似的表示距离相近（边界情况除外），这样可以利用字符串的前缀匹配来查询附近的POI信息



为什么分别给经度和维度编码？为什么需要将经纬度两串编码交叉组合成一串编码？

我们将二进制编码的结果填写到空间中，当将空间划分为四块时候，编码的顺序分别是左下角00，左上角01，右下脚10，右上角11，也就是类似于Z的曲线，当我们递归的将各个块分解成更小的子块时，编码的顺序是自相似的（分形），每一个子快也形成Z曲线，这种类型的曲线被称为Peano空间填充曲线。这种类型的空间填充曲线的优点是将二维空间转换成一维曲线（事实上是分形维），对大部分而言，编码相似的距离也相近， 但Peano空间填充曲线最大的缺点就是突变性，有些编码相邻但距离却相差很远，比如0111与1000，编码是相邻的，但距离相差很大。

解决的思路很简单，我们查询时，除了使用定位点的GeoHash编码进行匹配外，还使用周围8个区域的GeoHash编码，这样可以避免这个问题。

我们已经知道现有的GeoHash算法使用的是Peano空间填充曲线，这种曲线会产生突变，造成了编码虽然相似但距离可能相差很大的问题，因此在查询附近餐馆时候，首先筛选GeoHash编码相似的POI点（即使相似的点，可能隔得很远，因为编码是连续的），然后进行实际距离计算。geohash只是空间索引的一种方式，特别适合点数据，而对线、面数据采用R树索引更有优势

除Peano空间填充曲线外，还有很多空间填充曲线，如图所示，其中效果公认较好是Hilbert空间填充曲线，相较于Peano曲线而言，Hilbert曲线没有较大的突变。为什么GeoHash不选择Hilbert空间填充曲线呢？可能是Peano曲线思路以及计算上比较简单吧，事实上，Peano曲线就是一种四叉树线性编码方式

The main advantage in using **space filling curves** is that they map points from higher dimensional space to space of lower dimension. For example, they make it possible to window query for points using traditional B-tree database index.

Google S2 and Hilbert space filling curve: <http://blog.christianperone.com/2015/08/googles-s2-geometry-on-the-sphere-cells-and-hilbert-curve/>

Since S2 uses the Hilbert Curve to enumerate the cells, this means that cell values close in value are also spatially close to each other. When this idea is combined with the hierarchical decomposition, you have a very fast framework for indexing and for query operations. The cells are represented in 64-bit integers.

基于R树的附件POI搜索： <http://blog.csdn.net/v_july_v/article/details/11288807>

R树是B树在高维空间的扩展，是一棵平衡树。每个R树的叶子结点包含了多个指向不同数据的指针，这些数据可以是存放在硬盘中的，也可以是存在内存中。根据R树的这种数据结构，当我们需要进行一个高维空间查询时，我们只需要遍历少数几个叶子结点所包含的指针，查看这些指针指向的数据是否满足要求即可。这种方式使我们不必遍历所有数据即可获得答案，效率显著提高.R树运用了空间分割的理念，这种理念是如何实现的呢？R树采用了一种称为MBR(Minimal Bounding Rectangle)的方法，在此我把它译作“最小边界矩形”。从叶子结点开始用矩形（rectangle）将空间框起来，结点越往上，框住的空间就越大，以此对空间进行分割。

首先我们假设所有数据都是二维空间下的点，图中仅仅标志了R8区域中的数据，也就是那个shape of data object。别把那一块不规则图形看成一个数据，我们把它看作是多个数据围成的一个区域。为了实现R树结构，我们用一个最小边界矩形恰好框住这个不规则区域，这样，我们就构造出了一个区域：R8。R8的特点很明显，就是正正好好框住所有在此区域中的数据。其他实线包围住的区域，如R9，R10，R12等都是同样的道理。这样一来，我们一共得到了12个最最基本的最小矩形。这些矩形都将被存储在子结点中。

下一步操作就是进行高一层次的处理。我们发现R8，R9，R10三个矩形距离最为靠近，因此就可以用一个更大的矩形R3恰好框住这3个矩形。

同样道理，R15，R16被R6恰好框住，R11，R12被R4恰好框住，等等。所有最基本的最小边界矩形被框入更大的矩形中之后，再次迭代，用更大的框去框住这些矩形。

用地图的例子来解释，就是所有的数据都是餐厅所对应的地点，先把相邻的餐厅划分到同一块区域，划分好所有餐厅之后，再把邻近的区域划分到更大的区域，划分完毕后再次进行更高层次的划分，直到划分到只剩下两个最大的区域为止。要查找的时候就方便了。

下面就可以把这些大大小小的矩形存入我们的R树中去了。根结点存放的是两个最大的矩形，这两个最大的矩形框住了所有的剩余的矩形，当然也就框住了所有的数据。下一层的结点存放了次大的矩形，这些矩形缩小了范围。每个叶子结点都是存放的最小的矩形，这些矩形中可能包含有n个数据。

地图查找的实例

讲完了基本的数据结构，我们来讲个实例，如何查询特定的数据。又以餐厅为例，假设我要查询广州市天河区天河城附近一公里的所有餐厅地址怎么办？

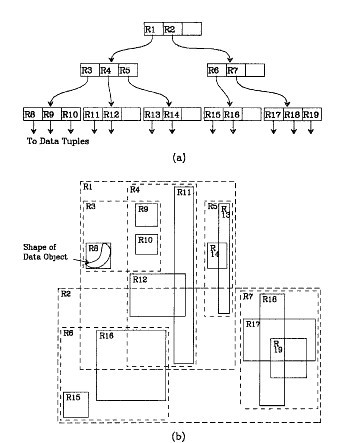
打开地图（也就是整个R树），先选择国内还是国外（也就是根结点）；

然后选择华南地区（对应第一层结点），选择广州市（对应第二层结点），

再选择天河区（对应第三层结点）；

最后选择天河城所在的那个区域（对应叶子结点，存放有最小矩形）；

单纯用R树来作索引，搜索附近的地点，可能会遍历树的很多个分支。而且当全国的地图或者全省的地图时候，树的叶节点数目很多，树的深度也会是一个问题。一般会把地理位置上附近的节点（二维地图中点线面）预处理成page(大小为4K的倍数)，在这些page上建立R树的索引



叶子结点所保存的数据形式为：(I, **tuple-identifier**).其中，tuple-identifier表示的是一个存放于数据库中的tuple，也就是一条记录，它是n维的。I是一个n维空间的矩形，并可以恰好框住这个叶子结点中所有记录代表的n维空间中的点。I=(I0,I1,…,In-1)

非叶子结点的结构其实与叶子结点非常类似。想象一下B树就知道了，B树的叶子结点存放的是真实存在的数据，而非叶子结点存放的是这些数据的“边界”，或者说也算是一种索引.同样道理，R树的非叶子结点存放的数据结构为：(I, **child-pointer**).其中，child-pointer是指向孩子结点的指针，I是覆盖所有孩子结点对应矩形的矩形

R-Tree与Geohashing的比较：<https://gis.stackexchange.com/questions/108557/advantages-of-r-trees-in-comparison-to-geohashes>

Geohash is very simple and effective way of indexing spatial features, particularly point features. Line and polygon features are little harder to index, but can be done. Geohash is a static hierarchical fixed size grid, overlayed on top of the earth surface. Grid cells of the same hierarchical level do not overlap. R-Tree is a dynamic grid which cell location and size change depending on the features they are indexing. R-Tree indexes features bounding boxes and cells change every time you insert and update data. Geohash is mostly used for indexing point features and cells **do not change** with every insert and update of data. Geohash cells do not adopt to the features like with R-tree.

Some of the advantages of geohash (comparing to r-tree) could be:

easy implementation

no performance degradation with growing number of features

proximity searches (partially true)

Some of the disadvantages of geohash (comparing to r-tree) could be:

arbitrary precision of grid. Geohash converts longitude and latitude coordinate into the one-dimensional string. Length of this string is directly tied to the converted precision of the coordinate

harder to index (and query) line and polygon features

size of the index could be large with some methods of line and polygon indexing

by the specifications, it can be only used with longitude/latitude coordinate system, although the same method could be applied to other coordinate systems also

Lucene, mongodb, Elasticsearch use geohash because geohash is mainly used for indexing points and there are lot of applications that need such a feature. Lines and polygons are not that often used (except for the GIS applications of course), so why bother with it. Other reason, is of course, ease of implementation. Geohash converts two-dimensional coordinate to one-dimensional value. This is called **dimensional reduction**. One-dimensional value is easy to be indexed by standard b-tree which is mostly used in those products.

I have to mention that there are similar algorithms to geohash but most of them are proprietary and require licensing. Geohash is in public domain. This could be also the reason for such a large usage in the recent years.

Uber, Yelp, FourSquare, and Mongo Engine are all using Google S2. fast and efficient searching of spatial data.

The main reason for using Google's S2 library is for its support for spatial indexing. As you put more and more geospatial data in a map, your ability to query it in a timely and effective manner reduces dramatically. This is a problem for the likes of Uber, Foursquare and Yelp who deal with colossal amounts of spatial data.

The S2 library attempts to resolve this using a very clever construct called the Hilbert Curve. It's basically a curve that occupies a space, covering all the areas within that space. Now the reason this is cool is that it means you can plot map points on this curve and when you "stretch" out the curve into a long string and add an index to each entry, all those points that were close together on the map remain close together in terms of their index position.This is great news for searching, because it's a lot easier to find points that are near to each other on a map when you can access them via their indexes and they are still close together. It means you don't have to scan the whole list of points to find ones that are geographically close, but just a few either side of where your starting position is.

CDN原理 <http://grefr.iteye.com/blog/2004248>  
CDN本质和反向代理一样是load balancer，核心是用cache，只不过CDN的机器也可以push数据，也可以pull数据

CDN，全称Content Delivery Network，主要作用是为源站减少访问压力的同时，为客户端提供更快速的内容响应。除此之外，CDN还能对源站进行安全防护。 其实真正为CDN付费的是源站，所以CDN的用户其实是源站，例如新浪微博，youku视频，淘宝网啊之类的。而客户端，是CDN的用户的用户。 所以CDN是夹在源站和源站的用户之间的

DNS的工作过程容易被人忽略，一般只知道DNS的输入是一个网址，输出的是一个IP。 DNS的协议主要是基于UDP的，所以dns server的qps一般都是很惊人的，比web server（http是基于tcp的）的qps是高出几个量级的。有个基本概念就是dns的记录类型，常见的dns记录类型有A，AAAA，CNAME等。中A记录是域名到IPV4地址的；AAAA记录是域名到IPV6地址的；CNAME记录类似于查询过程中的转发，意思是你去问问这个人，他管这事。好的，下面继续说说DNS的工作过程。

1.在浏览器键入www.taobao.com,其实真正dns协议里用到的是www.taobao.com.最后还有一个点，可能是因为美观等原因，一般都不显示

2.查询本地缓存（host文件或者是浏览器的缓存）中有没有该域名对应的记录，有的话就直接用了

3.向运营商的DNS服务器发起dns解析的请求，一般称运营商的DNS服务器为local dns

4.local dns会查询本地的缓存，local dns设置的缓存时间是有讲究的，过长过短都不好。另外local dns的查询是运营商的事，外部不可控

5.local dns如果没有缓存，会把域名从右往左扫描，依次请求对应的服务器，例如对于域名www.taobao.com.，先去问负责.的根域名服务器，就是传说中全球只有几台的那些服务器，他们会答复.com是谁管理的，然后local dns又去找管理.com的服务器（假设名字为S1），去问问taobao.com是谁管，一般来说，在S1查到的记录是一条cname记录（阿里毕竟大公司，自己管理自己旗下的域名），然后就转到了阿里自己的DNS服务器上来了，一般称之为权威服务器

6.权威服务器是阿里自己建的，然后根据公司内部的一些配置啊，调整啊，查到www.taobao.com.对应的服务器是谁，返回一个IP地址

7.local dns缓存这个IP地址，并且回复浏览器

8.浏览器和对应的IP地址的服务器简历TCP连接，发送HTTP报文

CDN访问过程(使用缓存服务)

CDN网络是在用户和服务器之间增加Cache层，主要是通过接管DNS实现,将用户的请求引导到Cache上获得源服务器的数据

下面让我们看看访问使用CDN缓存后的网站的过程：

1.用户输入访问的域名,操作系统向 LocalDns 查询域名的ip地址.

2.LocalDns向 ROOT DNS 查询域名的授权服务器(这里假设LocalDns缓存过期)

3.ROOT DNS将域名授权dns记录回应给 LocalDns

4.LocalDns得到域名的授权dns记录后,继续向域名授权dns查询域名的ip地址

5.域名授权dns 查询域名记录后(一般是CNAME)，回应给 LocalDns

6.LocalDns 得到域名记录后,向智能调度DNS查询域名的ip地址

7.智能调度DNS 根据一定的算法和策略(比如静态拓扑，容量等),将最适合的CDN节点ip地址回应给 LocalDns

8.LocalDns 将得到的域名ip地址，回应给用户端。用户得到域名ip地址后，访问站点服务器

9.CDN节点服务器应答请求，将内容返回给客户端.(缓存服务器一方面在本地进行保存，以备以后使用，二方面把获取的数据返回给客户端，完成数据服务过程)

通过以上的分析我们可以得到，为了实现对普通用户透明(使用缓存后用户客户端无需进行任何设置)访问，需要使用DNS(域名解析)来引导用户来访问Cache服务器，以实现透明的加速服务. 由于用户访问网站的第一步就是 域名解析 ,所以通过修改dns来引导用户访问是最简单有效的方式.

CDN网络的组成要素

对于普通的Internet用户，每个CDN节点就相当于一个放置在它周围的网站服务器.

通过对dns的接管，用户的请求被透明地指向离他最近的节点，节点中CDN服务器会像网站的原始服务器一样，响应用户的请求.由于它离用户更近，因而响应时间必然更快.

智能调度DNS(比如f5的3DNS)

智能调度DNS是CDN服务中的关键系统.当用户访问加入CDN服务的网站时，域名解析请求将最终由 智能调度DNS 负责处理.

它通过一组预先定义好的策略，将当时最接近用户的节点地址提供给用户，使用户可以得到快速的服务.

同时它需要与分布在各地的CDN节点保持通信，跟踪各节点的健康状态,容量等，确保将用户的请求分配到就近可用的节点上.

缓存功能服务

负载均衡设备(如lvs,F5的BIG/IP)

内容Cache服务器(如squid）

共享存储(根据缓存数据量多少决定是否需要)

Consistent hashing 数据迁移时如何同时服务用户请求呢? 数据迁移的时候，这个迁移你不要理解为 move。要理解为 migration。这个过程分为：

1.数据的复制。数据从机器A到机器B。机器A的数据依旧存在，依旧服务用户

2.数据的验证。比较A和B机器的数据的一致性。不一致的地方进行修复。此时依旧是A服务用户

3.数据的切换。切断A的服务，改为B服务。

在作 consistent hash 的时候通常一台机器是对应好几个 virtual nodes. 然后把这些 nodes 随机撒在环上. 在作 replicas 时, 通常也是拿环上顺时钟方向的下三个 nodes (given RF=3) 来当 replicas.

怎么避免 virtual nodes 被撒在环上的连续空间.

譬如一台机器有 5 个 virtual nodes. A => A1, A2, ... , A5

如果他们刚好被放在环上的连续空间, 作 replicas 的时候, 其实都作在同一台机器上, 一台掛了就所有 replicas 一起掛.

这个问题假设，是不是同一台机器的node在连续的区间里面？ 那这样是不是重新构造一下随机函数就解决了？就是先随机选机器， 第二部，在随机选这台机器上面的node.

In consistent hashing a node is responsible for keys with ids from itself to its successor. Consider what happens when a node fails. Let's call this node X. All keys for which X was was responsible are now assigned to Y. The effect of X failing has not been distributed across the system but concentrated on a single node.(Here X and Y are physical nodes)

Now assume we have virtual nodes X1, X2, ... Xn corresponding to node X. Let Y1, Y2, ... Yn be the predecessors(逆时针) of these virtual nodes. When X fails, the keys for which X was responsible (i.e. keys for which X1, X2, ... Xn were responsible) have been distributed across virtual nodes Y1, Y2, ... Yn (which may not correspond to the same physical node).

Similarly, when a new node is added to the system, using virtual nodes will ensure that the new node soaks load from many other physical nodes rather than splitting load with just one other physical node.

Another advantage is that we can have different number of virtual nodes corresponding to each physical node depending on the node's computational power. More virtual nodes implies that the node will get responsibility of more keys which allows us to handle heterogeneity of nodes gracefully.

Uber pool: <https://www.quora.com/How-do-the-UberPOOL-and-Lyft-Line-matching-algorithms-compare>

The key variables to play with are: routes, # of passengers, wait time, price, matching.

Routes: Lyft introduced their 'hot spot' to their passengers - what it means is -- if you can walk couple blocks to one of their hot spots then the ride is cheaper -- if Lyft manages to lure more passengers to be around the hot spot that would mean further increased efficiency i.e. the drivers don't have to go further away from the primary routes to pick passengers. This in turns means reduced time to pick up and generally overall quality of the experience. Anyway, Uber followed Lyft by introducing 'smart routes', their version of the 'hot spot'.

# of passengers in 1 request: Uber is more strict, while Lyft generally allows up to two passengers. Uber's pricing is slightly ($~1) cheap if it's 1 passenger vs. 2.

Wait time: Lyft introduced a 10min matching window, which allows more time for matching, hence better route optimization. In order to incentivize the passengers the 10min wait time as opposed to the regular 1min is a bit cheaper ~$1-2. For some reason, Uber hasn't followed this yet. Arguably, Uber thinks it's better for the overall passenger experience if the wait time is shorter.

Matching: This is perhaps the key piece. Lyft looks for nearby riders -- nearby could mean <=0.5mile radius or more and locks it when there's a match. Whereas Uber started the same way, but moved into a ongoing pull model i.e. even when the 1st passenger is matched with the 2nd, they could get a request for the 3rd and 4th passenger if the car is near by the pick up points. Again, there may be artificial inefficiencies for the reason that both Lyft/Uber may want to keep their drivers busy, hence they may prefer to pool requests for the 1st two passengers per car. But it would be in the best interest of the marketplace to pack maximum # of riders in one car, keep the wait time to be as low as possible, provide the service with the lowest price points, and keep the drop times to be as quick as possible.

News feed + Chat

最近的一个高频面经，说fb回复评论后，如何能不刷新页面而实时显示最新评论.这个就是我们最后一节课讲的聊天系统  
1.点开同一个帖子的人订阅到同一个channel.用户上线之后，自动订阅所属的channel。  
2.收到同一个channel的push,或者通过long polling拿到. 长连接和socket，都是用户打开app就连接上了，然后用户关闭app就断开了  
关键词 channel subscribe websocket

real-time push service 不push给不在线或没有subscribe channel的follower。当lady gaga发tweet, 需要1次db/cache read 来get 所有followers, 然后在内存判断哪些人要push并push?

一个人"在线"，就要占用掉一个端口，1B用户要占1B个端口，哪有这么多机器的端口给你占。并不是所有的人都会“在线”，不在线的用户就无法被push到了.

Group chat这一块，490个人不在线的时候，push server只向在线的10个人发消息。那等那490个人陆陆续续上线之后，push server会向这490人发送之前只向10个人发送的消息吗？不会陆续发送。没必要。这490个人会主动来pull的！

For chat application, usually you want to maintain persistent network connection between the client and the server. We can use Socket to provide a bi-directional communication channel between a client and a server. This means the server can push messages to clients. That way, client don't have to poll the server frequently for changes which can be a burden especially when there are many connected clients at one time.

When a person sends a message in the chat application, the server will get it and push it to all other connected clients. Seems like maintaining one million web sockets is not an issue on a single server. WebSockets are fundamentally long-lived TCP sockets with a HTTP-like handshake and minimal framing for messages.On today's systems, handling 1 million concurrent TCP connections is not an issue.We installed Lightstreamer Server (which is a **WebSocket server**, among other things) on a m2.4xlarge instance. This means 8 cores and 68.4 GiB memory.

Facebook online status: 每隔3~5秒上报自己的status，同时pull自己所有好友的状态. 那么这每隔3~5次发pull请求的服务器的压力不大吗? 因为这个是Real Time的数据流，使用redis集群来做，只要查询不穿透到DB上就行，所以压力不大. 这里的服务器是专门处理online status的服务器.跟Message server分开

WhatsApp online status用pull的时候，用户通过http来发heart beat，同时要一下好友的在线状态，那用户是跟web server还是跟push server通信？

跟 web server通信。push server是一个内部系统，不直接接受用户的请求。

web server是知道的，因为web server会把用户最后 发pull请求的时间存在db里。根据这个时间去判断用户在不在线（比如上一次发pull请求的时间在10s以内之类的）

这里不需要用到 Push server，也就是说，这里的“在线”状态不是看是否和 push server 连没连上，而是看你什么时候没给web server发pull request。

你可以试试看在fb创建一个用户，然后和你的账号加好友，然后你自己跟自己聊天。然后再把其中一个账户马上下线。看看另外一个账户是不是能够马上收到下线的通知。如果马上收到，说明走了 push server，如果不能（比如隔10s才看到下线），说明是 pull model。

**Push Notification**: Applications like Facebook, Quora, Twitter have notifications system, where any comment done by someone is received as a notification from server.

As far as I know most doable way is using Long Polling mechanism and using central pub/sub server like Redis. And for performance use some kind of non-blocking server.

Challenge

Lets say you wanna notify all the users when someone posted an answer on a question.

Steps

1.When someone press "Follow Question"

2.Use unique id of the question to create a new channel in redis.

3.Subscribe new user to the channel you created just now based on question id.

4.Add user(or some unique info about it) to redis set and check the set so that same user does not end up getting duplicate notifications.

5.When someone posts an answer, send notification to channel using question id.

6.Wrote a little javascript to long poll the server so that when new notification appear on the channel, it will directly be pushed to ui.

7.When someone unfollow the question just remove from set and unsubscribe from channel.

How does messaging work in WhatsApp?

WhatsApp or most of the other messaging apps rarely work on a peer to peer basis. So it wouldn't open a connection (from your device) to each of your friends' devices. Instead your device connects to their server. It could then use a custom TCP protocol or maybe HTTP to communicate your messages to the server. The server in return would dispatch them to your friends' devices. If your friend had their app open or at least the app process running there might be a live connection to the server. WhatsApp will use that connection to send them your messages. If their app is "offline" then they might choose to send them a push notification instead.

1.Alice decides to message Bob. Alice's phone establishes a connection to the WhatsApp server and it is established that this connection is definitely from Alice's phone. Alice now sends via TCP the following message: "For Bob: A giant monster is attacking the Golden Gate Bridge". One of the WhatsApp front end server deserializes this message and delivers this message to the actor called Alice.

2.Alice the actor decides to serialize this and store it in a file called "Alice's Sent Messages", stored on a replicated file system to prevent data loss due to unpredictable monster rampage. Alice the actor then decides to forward this message to Bob the actor by passing it a message "Msg1 from Alice: A giant monster is attacking the Golden Gate Bridge". Alice the actor can retry with exponential back-off till Bob the actor acknowledges receiving the message.

3.Bob the actor eventually receives the message from (2) and decides to store this message in a file called "Bob's Inbox". Once it has stored this message durably Bob the actor will acknowledge receiving the message by sending Alice the actor a message of it's own saying "I received Msg1". Alice the actor can now stop it's retry efforts. Bob the actor then checks to see if Bob's phone has an active connection to the server. It does and so Bob the actor streams this message to the device via TCP.

4.Bob sees this message and replies with "For Alice: Let's create giant robots to fight them". This is now received by Bob the actor as outlined in Step 1. Bob the actor then repeats Step 2 and 3 to make sure Alice eventually receives the idea that will save mankind.

WhatsApp uses XMPP (eXtensible Messaging and Presence Protocol) to handle the message delivery system.

XMPP is mostly like HTTP where the client opens the socket with the XMPP server and keeps it open as long as the client is logged in. It's not like the regular REST API where the client opens the socket send/receive the data and close the socket. The socket is open as long as you are signed in. In case of WhatsApp that's eternity (not really, WhatsApp reconnects automatically if the connection terminates) XMPP protocol has been used in various chat applications such as Google Talk, Facebook messenger etc.

<https://bitbucket.org/sco0ter/babbler>

XMPP is an application protocol while WebSockets is a transport protocol. Babbler is a young XMPP client library for Java.

A session to an XMPP server can be established in three ways (connection methods):

1.By a normal TCP socket connection

2.By a BOSH connection(Bidirectional-streams Over Synchronous HTTP, long-polling )

3.By a WebSocket connection

Channel里存的是，”当前“有哪些用户”在线“并且”订阅“到了对应的channel. 放到内存中就行，不需要persist到Disk

User和channel的mapping(Channel Table)需要有persistent storage,我在这个群的群聊当中的mapping(Message Table)，那个是存在 Message Service 中的。不需要Channel Service去管，Channel Service 只管在线的。如果内存挂了数据丢了，就大不了不给 push，用client 的fallback 机制（比如pull）来获得新消息，并且 client 在一段时间之后也能知道自己掉线了，会重新连接，这样 Channel Service 中的数据就能重新 build 起来。rebuild的方式是每次user上线message service都会 request channel service里把user重新add into all his/her channels/threads

当用户推出channel/thread时，如果你没掉线，你可以发送一条指令给web server（channel service），让web server 帮你unsubscribe。

如果你掉线了，push server会首先知道你掉线了（因为socket是双向的，一边掉了另外一边也就知道断开了），然后push server就可以通知 channel 你掉线了，把你unsubscribe 掉。

How to recover when a chunk is broken?

CS4 和 CS3，CS5之间是replica的关系（consistent hashing 顺序找的三个server，数据一式三份）  
CS4 通过checksum发现了自己的chunk3坏了，所以他需要recover，就得找master（因为master知道他的replica是谁，consistent hashing表在master上面).于是，CS4就向CS3或者CS5要chunk3(找距离近的，same rack, same room, same region)，进而修复自己的chunk3。

Large distributed system 怎么log 各种data analysis可能需要的各种query，比如怎么得到过去一周或一个月top 10 requests或者exceptions?

大概就是，首先各个机器先要log到自己的机器上，然后在自己的机器上做一次整合(Aggregate raw data)，整合之后，每隔一段时间集体发给一台中心节点存储起来。整合包括了，比如某个request在一秒钟之内发生了10次，没有必要记录了成10个requests

NoSQL 与 SQL:

为什么nosql database的qps要比sql的高. 因为存储结构简单，查询效率就高。存储结构简单是把双刃剑，查询效率高的同时，所支持的查询的多样性也就差了。极端一点，你看文件系统，直接读文件效率最高吧，但是他只支持 文件名->内容的这种查询，不支持什么 range query

NoSQL有分很多种,其中key-value NoSQL (Redis, MemcacheD, etc)的选用相对比较清楚些,大多是当后端Data storage的cache层来用.MongoDB is document based, 对于OOD特别方便，而且Query快。以下主要讨论Column Family NoSQL(eg Cassandra, Hbase)和SQL之间的选用

Column Family NoSQL 的schema 和SQL schema 大多能够互相作逻辑转换. 也就是说, 给一个DB, 里面有很多table, table 里有很多column, 然后跟你说我query 的型态会长怎样(等同告诉你app layer 的join 要怎么作), 我们多半能把这些DB schema 转成CF NoSQL 的schema. 反之亦然.对single box (单一机器) 来说, CF NoSQL 能承受的qps 比SQL 要高, 不过在multiple machines 的情况下, 可对SQL 去作sharding & replicas 来增加其performance 和availability/reliability. 这边甚至可混用consistent hashing 的架构来作SQL sharding/replication. 也就是说, 在多台机器可用的环境下, CF NoSQL 和SQL 的效能是可以作到差不多的.

1. Data非常不relational (require no join or few joins),这时用SQL就有点浪费,可能会有不必要的overhead.

2. Data非常relational (require lots of joins)或有大量的columns要作index,这时用CF NoSQL可能要处理大量的de-normalization,虽然disk便宜,但duplicated data太多的话可能也会爆容量?而且update时要处理de-norm data间consistency的问题. eg一个data可能属于(row\_key\_A, column\_key\_A)同时也属于(row\_key\_B, column\_key\_B),这样更新这data时就要同时更新这两个row. 感觉这种情况选用SQL会较佳

3.去除以上两个极端cases,通常data是介于中间.这时候感觉用CF NoSQL和SQL是差不多的.

用SQL的话, developer要自己处理sharding/replication.不过相对而言, SQL expert的数量是远大于Cassandra/Hbase expert, SQL communities也相对成熟许多.

实现前5分钟，1小时，24小时内分享最多的post的系统 <https://www.jiuzhang.com/qa/219/>

server集群中过去24小时的top K exception

Given a continuous twitter feed, design an algorithm to return the 100 most frequent words used at this minute, this hour and this day. <http://stackoverflow.com/a/10190836/404145>

You need an array of 1440 (24\*60) word+count hash maps; these are your minute-by-minute counts<Post, Count>. You need two additional hash maps - for the rolling total of the hour and the day.

Define two operations on hash maps - add and subtract, with the semantic of merging counts of identical words, and removing words when their count drops to zero.

Each minute you start a new hash map, and update counts from the feed. At the end of the minute, you place that hash map into the array for the current minute, add it to the rolling total for the hour and for the day, and then subtract the hash map of an hour ago from the hourly running total, and subtract the hash map of 24 hours ago from the daily running total.

Finally, you need a way to produce the top 100 words given a hash map. This should be a trivial task - add items to an array of word+count entries, sort on the count, and keep the top 100. A better way, to actually compute the top K words given a min/hour/day, it is faster to use partition (O(n)), quick select, rather than sorting (O(nlgn))

从算法的角度分析, LFU,但是啥时候remove expired呢，query的时候就行吧

从算法的角度，可以简单的称之为 Top K Frequent Elements in Recent X mins.

算法的角度，本质就是设计一个数据结构，支持给某个key的count+1（有一个post被分享了），给某个key的count-1（有一个分享的计数已经过期了），然后查询Top k。

做法是维护一个有序序列（用链表来维护），每个链表的节点的key是 count，value是list of elements that has this count，也用linked list串起来。 比如 a出现2次，b出现3次，c出现2次，d出现1次。那么这个链表就是：{3: [b]} --> {2: [a ->c]} --> {1: [d]} 然后另外还需要一个hashmap，key是element，value是这个element在链表上的具体位置。因为每一次的操作都是 count + 1和 count - 1，那么每次你通过 hashmap 找到对应的element在数据结构中的位置，+1的话，就是往头移动一格，-1的话，就是往尾巴移动一格。总而言之复杂度都是 O(1)。当你需要找 Top K 的时候，也是 O(k)的时间 可以解决的。

从系统设计的角度分析

一般来说，你可能首先需要按照 LFU 的思路答出上述的方法。这个就过了第一关，算法关。但是还没结束，这个题还有第二关，那就是系统设计关。上面的算法从算法的角度没办法更优了，每个分享操作都是O(1)的代价，每个求Top K都是O(k)的代价。已经很棒了。但是系统的角度出发，会存在这样一些问题：

如果QPS比较高，比如 1M，这个数据结构因为要加锁才能处理，所以会很慢。分享的数据本身是分布式的，而不是中心化的，也就是说，比如有1000台web服务器，那么这1000台web服务器的是最先获得哪个帖子被分享的数据的，但是这些数据又都分布在这1000台web服务器中，如果用一个中心化的节点来做这个数据结构的服务，那么很显然这个中心节点会成为瓶颈。

比如这个系统用在twitter 这样的服务中，根据长尾理论，有80%或者更多的帖子连 Top 20% 都排不进去。而通常来说，从产品的角度，我们可能只需要知道 Top 20, 最多 Top 100 的数据就可以了。整个系统浪费了很多时间去统计那些永远不会成为Top 100的数据。题目的要求是“5分钟，1小时，24小时”，而不是“最近2分零30秒”，“最近31秒”，也存在较大的优化空间. 真实产品实时性要求和准确性没有那么高。你需要查询最近5分钟的Top K，结果得出的是最近5分02秒的Top K在产品中是没有太大问题的。查询Top k 的次数远低于count + 1和 count -1 的次数。

综上所述我们给出一些针对性优化策略：

**分布式统计 Distributed**: 每隔5~10秒向中心节点(下边有解释)汇报数据

也就是说，哪些帖子被分享了多少次这些数据，首先在 web server 中进行一次缓存，也就是说web server的一个进程接收到一个分享的请求之后，比如 tweet\_id=100 的tweet被分享了。那么他把这个数据先汇报给web server上跑着的 agent 进程，这个agent进程在机器刚启动的时候，就会一直运行着，他接受在台web server上跑着的若干个web 进程(process) 发过来的 count+1 请求。

这个agent整理好这些数据之后，每隔5~10秒汇报给中心节点。这样子通过5~10s的数据延迟，解决了中心节点访问频率过高的问题。这个设计的思路在业界是非常常用的（做这种数据统计服务的都是这么做的），我们在《系统设计班》的datadog一节的课中，就讲到过用这种思路来统计每一个event发生了多少次。

**分阶段统计 Level**

在《系统设计班》的 ratelimiter 一节课中，我们也提到了这种分阶段统计的思想。即如果我要去算最近5分钟的数据，我就按照1秒钟为一个bucket的单位，收集最近300个buckets里的数据。如果是统计最近1小时的数据，那么就以1分钟为单位，收集最近60个Buckets的数据，如果是最近1天，那么就以小时为单位，收集最近24小时的数据。那么也就是说，当来了一个某个帖子被分享了1次的数据的时候，这条数据被会分别存放在当前时间(以秒为单位），当前分钟，当前小时的三个buckets里，用于服务之后最近5分钟，最近1小时和最近24小时的数据统计。

你可能会疑惑，为什么要这么做呢？这么做有什么好处呢？这样做的好处是，比如你统计最近1小时的数据的时候，就可以随着时间的推移，每次增加当前分钟的所有数据的统计，然后扔掉一小时里最早的1分钟里的所有数据。这样子就不用真的一个一个的+1或者-1了，而是整体的 +X 和 -X。当然，这样做之后，前面的算法部分提出来的数据结构就不work了，但是可以结合下面提到的数据抽样的方法，来减小所有候选 key 的数目，然后用普通的 Top K 的算法来解决问题。

**数据抽样 Sample**

可以进行一定程度的抽样，因为那些Top K 的post，一定是被分享了很多很多次的，所以可以进行抽样记录。如果是5分钟以内的数据，就不抽样，全记录。如果是最近1小时，就可以按照比如 1/100 的概率进行 sample。这个思想我们在Web Crawler的那节课中提到过。We need only top K, we dont care about the real count.

**缓存 Cache**

对于最近5分钟的结果，每隔5s才更新一次。

对于最近1小时的结果，每隔1分钟更新一次。

对于最近24小时的结果，每隔10分钟才更新一次。

用户需要看结果的时候，永远看的是 Cache 里的结果。另外用一个进程按照上面的更新频率去逐渐更新Cache。提前把数据准备好

一个中心server来收集1000个web server的aggregated log, 这个中心server会single point of failure吗？

1000台机器，每台机器每隔10秒钟汇报一次，那么这台机器的QPS是100。这样的QPS一般不会搞挂机器。用slave或者double master之类的方式，我觉得在这种应用的情况下不是一个最优的选择。

要解决这个问题，我们必须明白两件事情：

1. 首先这个中心节点做什么？存数据么？

这个中心节点的任务是，接受来自1000台web server的统计数据，进行整理，并存储数据库中。

所以这台机器不是数据库。数据库是另外的专门的集群。这条机器只负责做一些整理工作，把一些有用的信息放到数据库和缓存里。方便你查询Top K 的时候用。他自己并不存储数据，只负责计算。

2. 中心节点挂了之后严重么？

不是那么严重。首先这个节点不直接服务普通用户，也就是说用户的request去的仍然是1000台web servers。这1000台web servers回去 cache 或者db里要数据，不会问这个中心节点要数据。所以他挂了，不会直接影响用户的感受。但是会间接影响数据的实时性和准确性。那么怎么解决这个问题呢？通常的做法是，用一个监控系统，监控这台机器是不是或者是不是正常工作，如果发现这台机器挂了，就发一个命令重启这台机器。通常来说更轻量级的做法是，在这台机器上有一个监控程序（比如supervisor），监控着主程序是不是正常执行中，如果挂了重启就行。这样已经足够用了。没有必要分摊成两台中心节点之类的，这样数据不统一在一台机器处理还会导致一些其他的维护麻烦的问题，得不偿失。

Design API,什么是data的serialzation/deserialzation?

数据在服务器之间/客户端与服务器之间通信，比较简单的是用http协议的形式，这样直接沿用web server的代码，比较方便。但是缺点是，http的形式比较慢，一则是http的包整个比较大，二则是也没有太多数据的压缩在里面。Facebook采用的是Thrift来进行数据交互，你可以理解为类似JSON的格式，不过做了压缩，人眼是看不懂了。效率，安全性等都比http+json的形式要好。Thrift 是专门拿来做服务器之间通信（RPC）用的Google用的类似的东西叫protobuffer

首先思考一下分布式系统中的 RPC (Remote Procedure Call) 问题，一个完整的 RPC 模块需要可以分为三个层次

· 服务层（service）：RPC 接口定义与实现

· 协议层（protocol）：RPC 报文格式和数据编码格式

· 传输层（transport）：实现底层的通信（如 socket）以及系统相关的功能（如事件循环、多线程）

在实际的大型分布式系统中，不同的服务往往会使用不同的语言来实现，所以一般的 RPC 系统会提供一种跨语言的过程调用功能，比如一段用C++实现的客户端代码可以远程调用一个用 Java 实现的服务

Thrift's primary goal is to enable efficient and reliable communication across programming languages. but I think HTTP-RPC can also do that, web developer almost everyone knows how to work on http and it is easier to implement HTTP-RPC(json) than Thrift?

A few reasons other than speed:

1.Thrift generates the client and server code completely, including the data structures you are passing, so you don't have to deal with anything other than writing the handlers and invoking the client. and everything, including parameters and returns are automatically validated and parsed. so you are getting sanity checks on your data for free.

2.Thrift is more compact than HTTP, and can easily be extended to support things like encryption, compression, non blocking IO, etc.(Thrift performance is comparable to the best JSON libraries (jackson, protostuff), and serialized size is somewhat lower)

3.Thrift can be set up to use HTTP and JSON pretty easily if you want it (say if your client is somewhere on the internet and needs to pass firewalls)

4.Thrift supports persistent connections and avoids the continuous TCP and HTTP handshakes that HTTP incurs. (No. 4 can be easily mitigated by using keep-alive which all HTTP clients support)

Top 10 products:　<https://www.jiuzhang.com/qa/686/>

设计top 10 most page view product

follow up 1.

比如有dvd，显示page view最多的10个dvd

比如有book，显示page view最多的10个book

follow up 2.

另外，traffic访问量很高怎么办

follow up 3

如果要实现上个月page view最多的10个dvd怎么办

----------- Weak Hire ------------

product每一次被访问时，page\_view+1：

UPDATE product SET page\_views = page\_views + 1 WHERE id=product\_id

选择top10的时候：

SELECT \* FROM product ORDER BY page\_views DESC LIMIT 10;

follow up 1:

SELECT \* FROM product WHERE product.category=dvd ORDER BY page\_views DESC LIMIT 10;

product.category 上需要index

------------- Hire ---------------------

首先需要向面试官分析为什么上面的方法不好：

上面的这个query非常的慢。因为数据库首先查一下index，看看有哪些product是dvd，然后将所有product 按照page\_views 排序，选出前10个。

这里page\_views如果你做index的话，因为要不停的+1+1，所以耗费会很大。

然后分析哪些地方我们可以入手进行优化：

1. top 10 无需是实时的 top 10。

2. 每次page view都导致数据库的一个项目+1这个操作比较费时。（本来只是一个读，却导致了写）

3. Page views的具体views可以不精确，有丢失。

其次提出一个更好的方案：

1. 将page view的log下来，这些log可以进行一些初步的aggregate，在每一台服务器上单独做就好了。log的结构基本就是，<product\_id, date, category, page\_views>，对于同一天的同一个产品的page\_views，我们可以把多条log aggrerate到一起。比如：<1, 2016-01-01, "dvd", 1> 和 <1, 2016-01-01, "dvd", 2> 可以aggregate为：<1, 2016-01-01, "dvd", 3>。这些log可以存在NoSQL的DB里。

2. 写一个 topk 的统计服务，针对每个类别进行 topk 统计。这里可以用一些 top k 的算法。网上 Google 一下有挺多（top k frequent elements）。简单的办法可以用Hash + Heap的方法。

follow up 2: 流量大的话，首先不能让每次page\_view都往数据库里写，所以log写到db之前，可以先写在内存里，进行aggregate，内存里的数据每隔几秒钟或者几分钟再往数据库里集体写一次。这样就减少了数据库的写操作次数。其次的话，每个类别的top 10，肯定是cache起来，然后加速读top 10的访问。

follow up 3: 因为存储的log就是按照时间存的，所以就解决了

A step-by-step design UberX following SNAKE.

Scenarios:

1: A driver log in/out/register/remove/update its profile in UberX

2: A driver starts its service and drive around his planned area. (The driver should see his/her locations moving on map)

3: A driver accepts/denies a rider's trip request.

4: A driver takes a rider to the destination. The fee should be updated frequently during the process.

5: When the trip ends, a driver shows the rider the fee and print the receipt for the rider.

6: When the trip ends, a driver rates the rider.

7. A driver checks his service statement and histories.

8: A rider log in/out/register/remove/update its profile in UberX.

9: A rider searches nearby cabs.

10: A rider posts a request.

11: A rider accepts/denies a cab.

12: A rider takes the cab to the destination.

13: A rider pay the fee and get the receipt.

14: When the trip ends, a rider rates the driver.

15: A rider checks his rider histories.

Necessary and Constraints:

Assumed number of full-time UberX Drivers: 1 Million

Assumed number of part-time UberX Drivers: 1 Million

Assumed number of Drivers during a peak hour: 1 Million \* 80% + 1 Million \* 20% = 1 Million

Assumed number of Drivers during a peak second: 1 Million \* 80% + 1 Million \* 20% = 1 Million

Assumed Average Trip Length: 20 minutes

Assumed Number of Trips during a Peak hour: 2 Million trips

Assumed Number of Trips during a Day: 2 Million trips \* 24 / 3 = 16 Million trips

Assumed Peak Trip Requests in a second: 2 Million trips / 3600 \* 2 = 1111 trip/second

Assumed Memory Requirements: 2 Million \* 20 / 60 \* 1 Kb = 667 Mb

Assumed Data Storage Requirements Per peak hour: 2 Million trips \* 1kb = 2 Gb

Assumed number of registered riders: 100 Million

Assumed Number of active riders during a Peak hour: 100 Million \* 0.1 = 10 Million

Assumed Number of active riders during a Peak second: 100 Million \* 0.1 \* 0.3 = 3 Million

Assumed Map Traffic Requirements: (3 Million + 1 Million) \* 10 Kb/s = 40 Gb/s

Application

1. UberX Map Service (real-time Demand and Supply)

2. User Service

3. Request and Dispatch Service

4. Business Logic Service (Fee & Receipt)

5. Payment Service (3th Party)

6. Estimated Arrive Time Service (Internal)

7. History Check Service

Kilobit: Data

1. Drivers - 2 Million \* 1kb = 2 Gb - Database - MySQL

2. Riders - 100 Million \* 1kb = 100 Gb - Database - MySQL

3. Request/Dispatch/Trip records During a Year - 16 Million \* 1 kb \* 365 - 11.68 Tb - NOSQL

4. Vehicle Location Histrory During a Year - 16 Million \* 1000 (points) \* 4Byte \* 3 (Lat/Lon/Direction) \* 365 = 70.08 Tb - NOSQL

5. Payment Histroy - 16 Million \* 1 kb \* 365 - 11.68 Tb - Distributed Database

Key Algorithms (Perfromance and Avalibility)

Dispatching Algorithm (Minimizing Estimated Time of Arrival)

Arrive Time Estimation Algorithm

Restful API: <https://www.zhihu.com/question/28557115>

大家都知道"古代"网页是前端后端融在一起的，比如之前的PHP，JSP等。在之前的桌面时代问题不大，但是近年来移动互联网的发展，各种类型的Client层出不穷，RESTful可以通过一套统一的接口为 Web，iOS和Android提供服务。另外对于广大平台来说，比如Facebook platform，微博开放平台，微信公共平台等，它们不需要有显式的前端，只需要一套提供服务的接口，于是RESTful更是它们最好的选择

1.URL root: <https://example.org/api/v1>

2.API versioning:可以放在URL里面，也可以用HTTP的header：/api/v1/

3.URI使用名词而不是动词，且推荐用复数。

· GET /products : will return the list of all products

· POST /products : will add a product to the collection

· GET /products/4 : will retrieve product #4

· PATCH/PUT /products/4 : will update product #4

4.使用正确的HTTP Status Code表示访问状态, 200, 300, 400, 500

两种设计数据库表的模式，Normalization对应的数据属于干净非冗余型，而Denormalization则允许数据冗余或者同样的数据存储于多处。下面主要列出了Normalization的优点和缺点，

优点：

1、数据更新更迅速；

2、数据存储空见通常更小；

3、在查询时减少了distinct和group by的使用

缺点：

1、查询时可能需要设计多个表，增加了join的使用；

2、一些并要的值需要在每次查询时计算（比如各种率值）

综合以上的优缺点，Normalization模式比较适合的场合是更新操作比较频繁的应用，即write-heavy。

同样的Denormalization则适合于read-heavy。

**Facebook, design event reminder, refer to Google Calendar below**

Scenario:

用户注册，登陆功能

用户创建event，邀请好友参加event

到了event时间，系统通知user

Service

User Service:

Register()

Login(username, password)

Event Service:

AddEvent(UserId)

Remind(EventID)

LookupEvent(UserId)

Storage:

以sql为例：

User Table(UserId, username, email)

Event Table(EventId, event\_happen\_time, created\_by\_userid, event\_address, notification\_time)

Invitation Table(EventId, userId)

用户创建event时，event Table插入一条新纪录。Notification\_time 是到了这个时间，系统通知created\_by\_userid。当用户A邀请B,C 参加event，Invitation Table插入两条记录：(userBId, eventId3) (UserCId, eventId3).

Optimized：

首先，如果需要通知的event很多，怎么处理？

到了notification\_time 通知用户，是不是可以保存这个东西在内存里，然后排序，到时间在通过email方式通知用户？

还有，如果一个用户创建那种每星期1下午3点都有的event，应该怎么建table？

这是一个典型的 Producer Consumer 模型。除了 Event Table 以外，我们需要一个 **Notification Table**。这个 Notification Table 记录的就是需要在什么时间，通知谁，什么事情，这一次性通知的，还是循环通知，如果是循环通知，那么循环规则是什么。 Notification Table 按照需要通知的时间进行排序。

先说如果 event少的 work solution 是什么：用一个进程，每隔 5s 扫一下这个table，不是线性扫描，而是 select 一下有哪些 Notification 的时间 小于 了当前时间。然后把需要通知的一个个通知。

如果需要通知的event很多，怎么处理？

一个人干不完的活，多个人干。如果通知特别多的话，我们就 sharding 呗。如果我们不需要支持删除一条 Notication 的话，我们甚至可以随意分给多个机器就好了。但这个问题中，不容易发现的是，如果我取关了一个 Event，需要把我的 Notification 删除。所以我们 sharding 的时候，应该按照被提醒的用户的 user\_id 进行 sharding。这样才能在需要删除的时候，还能再找回来。

Sharding 完以后，每个负责发提醒任务的进程就专门负责一个 sharding block 里的数据的提醒。

如果一个用户创建那种每星期1下午3点都有的event，应该怎么建table？

在前面的 Notification Table 的定义中，我已经给出答案了。基本思想就是，一条 notification 记录着下一次需要提醒的时间是什么，然后当这个 notification 被提醒之后，如果是一次性的，就直接扔掉了，如果是循环的，那么就根据循环规则计算出下一次需要提醒的时间，然后重新丢回 Notification Table 中。

**Facebook, privacy setting**

1. Friend 可见， Friend of Friends 可见

2. 每个user可以自己设些group，把一些user加进每个Group，其他用户不知道他设的group是什么，怎么让group里面的人看到我发的状态？怎么存group？

Scenario:

用户注册，登陆

用户A发post时选择可以看到的人是public，friend or group。其他用户能不能看到用户A的post。

Service：

User Service：

Register()

Login()

Post Service:

createPost()

Push:一切流程还是和Twitter的设计差不多，只是在post twitter的时候，有一个选项是可以选择: everyone, group, self这样的选项，如果我选择group的话，肯定我这个用户在之前就创建好了一个Group，于是发送了Tweet调用了write API把内容写到了我的NewsFeed上，这时通过我的social graph和group里所包含的人，进行消息push，push给group里的人的timeline list上。

我需要一张表叫Group表，里面有userID，groupID，groupName, including(有谁谁谁）字段，这样我就可以通过Group表找到我要发送tweets的人了.

Pull:如果一个用户的group里面有100个人，或者说有3个group每个group有30个人，那么直接通过group去查询对关系型数据库的查询的压力是很大的。所以我们需要做sharding和缓存。提供一种思路：我们不存friend信息，我们可以在一个post里面直接存不让哪些user\_id看到，然后这个信息在post的时候通过异步service去放到数据库里面。这样就不会让用户感觉到延迟。

**Facebook, Language localization就是语言的国际化**

1, 发一条post可以变成多种语言。

2, 网页上的很多字符内容需要根据用户所处的地区来用当地语言显示，怎么实现这个功能。

Service:

PostService

CreatePost(LanguageId)

TranslateService

Translate(FromLanguageId, ToLanguageId)

1、发一条post变成多个语言，这个根据用户的关注数量进行不同的区分对待：如果用户是明星，那么提前翻译好所有语言的版本的post存在缓存里面；不然的话就等到有人获取的时候再当场翻译（可以不在服务器端做，提供一个translate service，让用户的浏览器去调用api翻译，减轻服务器压力）。

2、把所有的翻译完的内容都存在数据库中并缓存，这样当用户要看的时候直接从缓存拿翻译好的内容就可以了。

**设计Google/outlook calendar**

Scenario

create event

invite user

notify users at specific time

notify user periodically

Service

User Service. 负责管理用户数据。

Event Service. 负责管理用户创建的 Events

Notification Service。负责notify用户某个 event 到了

Storage

这个题考点并不在存在什么数据库里，所以应该不会追问。主要的考点会是问 Event 要怎么存来实现 notify user at specific time 和periodically.

Event 大概需要存

created\_by (谁创建的)

strategy (一次性的，还是周期的。如果是周期的，那么周期的策略是什么，每周，还是每周末，还是什么，可以用一个具体的结构化数据表示出来，并serialize成json之类的存起来）

next\_notify\_time（下一次什么时候提醒，注意这里是下一次什么时候提醒，而不是第一次什么时候提醒）

attendes: many to many 的 users（实际上会存成另外一张表，记录哪个用户参与了哪个event）

下面整理一个完整的流程：

create event 就是数据库的 create 呗。

invite user。就是在 manytomany 的表里加一条记录，顺便记录一个状态，这个用户接受了邀请没有。

notify user at specific time。

首先我们需要搞清楚，到底要 notify 什么。这里有这么几种可能性：

A. 手机上弹出一个 notification 提醒用户。这个非常简单，这个不用走服务器那边。只需要用户的手机每隔一段时间 pull 一下最新数据，然后将需要提醒的event，按照 next\_notify\_time 倒序，然后用一个regular的进程每分钟看一眼就行了。因为一个用户登陆了自己的账户之后，才会被提醒。

B. 给用户发一封邮件之类的进行提醒。这个就比较难了，因为需要通过服务器去纷发。做法可以是这样，首先我们需要一个 MessageQueue 来做提醒任务的Broker（也就是消息队列来存需要马上去提醒的任务），然后有一个进程不断扫描数据库里的已经到了需要提醒的时间点（保险起见，减去1分钟），但是还没有提醒的 Events。然后把这些events标记为正在提醒，丢给 MessageQueue，然后启动一些专门复杂发邮件的机器，订阅 MessageQueue，这些机器得到了任务之后，就去发邮件，并且把任务标记为已经完成。

notify user periodically

其实看起来有点难，实际不难，只要你在第3步中完成了提醒之后，根据strategy看一下这个event是现在就结束呢，还是还有下一次提醒。如果还有下一次提醒，计算好下一次的时间，然后存更新一下 event 的next\_notify\_time 就好了。

Scale

Scale要看面试官问你什么具体的问题你具体去答了。从我看来，如果是手机那边的提醒，其实没什么难的，因为是手机local自己提醒，所以只要定期 Pull 新数据就好了。这个很容易 scale。

主要难Scale 的是，如果是发邮件提醒的那种events，要从服务器端来处理。那么一个进程扫描整个数据库看看到时间的 events有哪些这个是很慢很慢的。所以需要多加机器，通过sharding分开处理。我的想法是按照 created\_by 的 user\_id 进行 sharding 来让 events存在不同的数据库里。然后弄若干个进程，每个进程负责一个 micro sharding （一个进程负责不了一整个数据库，所以可以按照 micro sharding 来分任务），然后每个进程 Regular 的扫描自己的 micro sharding 中到期的 events，然后创建提醒任务，丢给 MessageQueue。

然后你可能会问 MessageQueue 会不会是瓶颈，答案是一般不会，MessageQueue 的性能是很强的，而且很多 MessageQueue 的技术很成熟，可以有 双 master 之类的模式来保证自己不挂。另外就是 MessageQueue 也可以有很多个呀，你指定好一个规则让不同的 扫描进程发送给不同的 Message Queue 就好了。

如何设计calendar的视图呢？（参考如何设计excel）

如何订阅多个feed呢？（参考如何设计subscribe系统）

如何解决多平台的修改冲突呢？（参考如何合并购物车, git, google doc）

**Facebook, web crawler**

download all urls from 1000 hosts. Imagine all the urls are graph.

Requirement: Each host has bad internet connection among each other, Has to download url exacly once

1.Bad internet connection 我觉得可以加上retry的逻辑，重试若干次。

2.Has to download url exacly once 这个必须要在数据库里做好标记是否已经download，跟课上讲的类似

3.可以用分布式数据库，甚至是Nosql, 内存型数据库Redis这样的，检测起来相当快。

crawler 是一个 client，不是一个server，他不提供服务给别人访问。只听从 scheduler 的调遣。

scheduler 是调度者，那么他就负责调度，谁干什么事儿，数据怎么存，哪篇网页该被抓取了，等等。所以是 scheduler 问 Task Table（某个database） 要数据，然后把数据给 crawler 去抓取，crawler干完活再向 scheduler 汇报说，我干完了，再给我一些吧。然后scheduler就再从 task table里拿一下给 crawler

Web crawler之后把Inverted Index(key-value)信息存到Trie上，Trie通过Consistent Hashing(很多machine组成一个环)来partition.

当有新machine join或者leave这个ring时怎么处理？

New machine joins:

1.Create new virtual servers/nodes

2.Pull data(trie里的一些branches) from clock-wise neighbor nodes(还不能update环上信息)

3.Construct new trie on new nodes

4.Update ring information(这个information存在每个node。ring information的信息很少，跟机器数目成正比)

5.Delete the old data from where we pulled(Optional)

Machine leaves/fails:

前提是这个machine的virtual nodes已经replicate到顺时针的neighbors去了,要不然不知道丢失nodes的data是什么

1. Clock-wise neighbors 接受request which used to be to the left virtual nodes

2. Update the offline trie and construct a new trie in memory(不能直接在online trie上做)

3. Switch the offline to be online(可以先把offline的上线，大家平摊流量，然后再把 online的下线。就没有gap time了)

**Facebook mutual friends**

当你访问一个 Facebook 的个人页面的时候，Facebook 会告诉你，你和他的共同好友有哪些人。如何设计这个功能？

如果每次都去好友里求一次，复杂度会是O(N \* max(N,M)) N是这个用户的好友数，M是他的好友的好友数量，响应时间会比较慢，可以提升响应速度而牺牲一点精确度，因为这里的top 10只是大概就可以，差别一两名也影响不大。

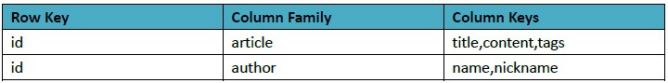
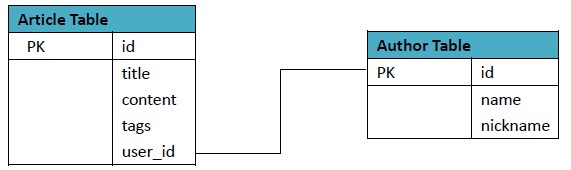
每个人存一个表，记录mutual friends 较多的top 10(可以存一下potential的100个popular好友)， 每次用户请求top 10时直接返回

当新加一个好友时，用异步服务器上述算法去求他俩的共同好友，并且与共同好友top 10表里面的最少共同好友的记录去比较，看是否需要更新结果。 同时，异步服务器也去更新他俩共同好友的共同好友表，因为他们因为这次操作多了一个共同好友。

每当一个用户去查看一个好友／聊天时，就发起异步服务器去求他俩的共同好友然后去更新他俩的表。（因为第一次加完好友以后，很可能他俩后来的共同好友有增加

HBase: <http://www.cnblogs.com/vajoy/p/5471308.html>

RDBMS 与 HBase Schema的对比



这里的 Column Keys 和 Column Family 可以看为这样的关系：

columId1 = { //id=1的行

article: { //ColumnFamily-article

title: XXX, //ColumnFamily-article下的key之一

content: XXX,

tags: XXX

},

author: { //ColumnFamily-author

name: XXX

nickname: XXX

}

}

HBash 的数据存储是基于列（ColumnFamily）的，且非常松散—— 不同于传统的关系型数据库（RDBMS），HBase 允许表下某行某列值为空时不做任何存储（也不占位），减少了空间占用也提高了读性能。HStore存储是HBase存储的核心，它由两部分组成，一部分是MemStore，一部分是StoreFiles。MemStore 是 Sorted Memory Buffer，用户写入的数据首先会放入MemStore，当MemStore满了以后会Flush成一个StoreFile（底层实现是HFile），当StoreFile文件数量增长到一定阈值，会触发Compact合并操作，将多个StoreFiles合并成一个StoreFile，合并过程中会进行版本合并和数据删除，因此可以看出HBase其实只有增加数据，所有的更新和删除操作都是在后续的compact过程中进行的，这使得用户的写操作只要进入内存中就可以立即返回，保证了HBase I/O的高性能。

**High Availability: 如何监控某个master是否还活着呢？**

建立一个HA系统，去ping这个master. HA 是高可用high availability机制的统称，我们拿简单的Codis具体，Codis是redis分布式的一个解决方法，那么他如何做到高可用，做到master宕机的时候恢复？在早期当中就是一个非常简单的HA tool这个tool ping master，一旦master挂了，就切换新的master

**针对同一条记录发生大量并发写的优化问题.**

1.Write back cache/Write through cache。大概的意思就是 Client 只负责写给cache，cache自己去负责写给数据库，但不是每条都写回数据库，隔一段时间写回去。

2.其实对同一条记录发生大量并发写的情况是很少的，即便Facebook这样的级别，最挑战的并不是短时间很多写，而是短时间很多读，因为只要是给人用的东西，一定是读多于写。短时间很多人读，而此时正好cache里又没有这个数据，这才是最头疼的。Facebook为了解决这个问题，拓展了memcached，增加了一个叫做 lease-get 的方法。

**How Facebook “lease get” solved Thundering Herd?**

A token is given to a client when that client experiences cache miss. Then, the following reads will have to wait until the cache have the data, instead of going to DB

以微博为例子，试想这样一个场景，一个大V发表了“中国足球队 1:0 战胜了韩国队”的一条帖子，然后很多微博上的人就来纷纷在这个帖子上留言，点赞，分享。就有很多人会在这个大V发了这个帖子之后的较短时间内，需要读取这个帖子的信息。短时间内的对于同一个帖子的QPS，可能会很高（比如 100K），这个行为就是一个“惊群”行为 - Thundering Herd。那么为了更好的响应用户的请求，就需要能有效的将这个数据 Cache 起来。因为 Cache 能够支持的 QPS会很高。

好，那么问题来了，假如这个时候，因为 Memory 不够了之类的原因，Cache Evicted 了这个帖子。也就是说，这 100K 的 cache 访问，通通会是 cache miss。那么我们通常cache和db的配合方式是什么？我们在课件上有写出来：

data = cache.get(key)

if data is None:

data = db.get(key)

return data

所以所有的 cache 访问会 fallback 给 db。那么我们知道db的承载力和 cache 相比是差几乎1000倍的。所以db就会一下子收到 100k 的 QPS。然后这个时候，db 的sharding机制是用不上的，因为这是去访问同一条数据，而不是不同的数据。所以一定是 hit 同一台数据库机器，那么这台数据库的机器就需要一下子承受 100k QPS 远超于他平时的量级的访问。这个时候db就会受不了，就会挂掉。

说回来，Lease get 之所以能解决“惊群”的问题，就是因为，他让你在data = cache.get(key) 的这一步卡一小会儿，让你等到数据被回填到 cache 之后，再返回，这样就不需要 fallback 回 db了，db也就不会挂了。leases work in practice because only one client is headed to the DB at any given time for a particular value in the cache.

**Thread table**. 在message system中 我们定义了thread table. 应该就是我的PPT上的Channel Table.

Thread table 格式为：

user\_id int64

thread\_id int64 create\_user\_id+timestamp

participant\_ids text json

participants\_hash string avoid duplicate threads

created\_at timestamp

updated\_at timestamp index= true

请问thread table表示的含义？对于两个人小刚和小王的对话， thread table 是否就是两个人的对话“窗口”？ 小刚和小王各有一个thread table？ 通过user\_id来区别？ participants\_hash是怎么避免重复的呢？

thread table 的含义：

首先先说一下 thread 的含义，thread就是你说的对话“窗口”，是两个对话人（甚至多个）共享的东西。比如参与者有哪些， 什么时候创建的，群聊名称是什么。

小刚和小王各有一个 thread table么？

是的。虽然是共享同一个thread，但实际上共享的数据有限，只有thread\_id，participants，thread\_nickname 等等。也有一些数据是私有的，比如 muted，unread\_count。所以他们需要各自有一个thread\_table。通过 owner\_id 来区分。因此 thread\_table 真正的 primary key 是：owner\_id + thread\_id

participants\_hash 如何避免重复？

participants避免的重复是产品角度的重复，即不允许同样的三个人创建多个只有这三个人的群。

**Facebook, advertisement system**

Desgin an Advertisement (AD) statistic system. 每次用户登录的时候，系统都会show几个广告给他。广告总共有K种类别，可以认为K<=10。用户看到广告可能会点击，Click Through Ratio (CTR) = 用户点的广告数量/给用户看的广告数量。注意若同一个广告被用户点击了多次，只算一个click。设计一个系统，answer the following two types of queries:

1. Given a user, return his CTR for all types of ADs.

2. Given an AD type, return its average CTR ovre all the users.

Follow ups:

a. What if K becomes very large? for example, we consider each product as one type, thus K can be as large as 10000.

b. New query type: Given an AD type, return the top-X users with highest CTR. 1< X < 100.

根据4s分析法来分析，首先确定出这是一个写qps很高的应用，然后再判断出follow up中的b问题如果直接去做query的话，肯定耗时太久。

那么首先回答delayed write caching的意思，就是说我的写入到数据库不是同步的，不是说传来一个用户的ctr我就写进数据库，是我先把一段时期的ctr全部缓存在内存，然后过了一定周期统计完一起写入到数据库，这样可以减轻数据库压力（缓存做好replica，异地三台保证数据不丢失）

**Hangman**

1. 要求用户可以输钱，赢钱  
2. 用户增加到5M +

Scenario:  
用户登录玩 hangman,  
5M user  
QPS ：来源于用户登录/充值，lookup 操作，start game, store game。

service?  
user service : register/login/look up(check balance, game history)/ update profile/buy god(充值)

word dict （根据难度分组，easy, medium, hard， 或者level1-10）

load game (different style game ?)  
根据用户选择的难度，load game. 然后开始。

如果有5个Million的game，要根据用户的情况生成不同的session?  
如果用户过多，我就多产生一些page, 例如   
hangman.com/game1-session1  
hangman.com/game1-session2  
hangman.com/game2-session12  
hangman.com/game1-session13

如何存储？  
实际上就是一个serialize 和 deserialize 的过程，存储当前游戏的状态，到了哪一步，还有多少次猜的机会，还有什么character available.

如果用户掉线？  
如果游戏还没完，检测到用户掉线，就把当前游戏存进DB（因为涉及到输钱赢钱的问题，不可能用户看快输了就拔线），以及想到此处可以和面试官讨论并且用memcache优化。

如果是自己玩的话就按照地区sharding，如果是多人玩的话，有两种可能：不同国家（地区）的不在一起玩，就像现在大型网游做的那样，分不同的地区服务器，其实就是按照地区进行sharding；全球同服，那么就在玩家数最多的三个地区放最多的webserver，然后采用delay write，把数据放到缓存里去解决访问速度、性能和高qps的问题。

**Sharding key**：

错误的理解：关于数据horizontal Sharding，对于MySQL我理解应该是按照主键进行Sharding，此时很难再给其它非主键增加索引了吧？也就是说对于MySQL, Sharding一般只用于检索主键的场景吧？

这个理解并不对。我举个例子，比如你存用户的基本信息。然后比如你按照 user\_id 进行Sharding。这个时候，比如对于用户登录来说，有一个需求是根据 username / email 去找到对应的 user 的信息。那么就需要用一个其他的表/table，来存储 username / email => user\_id 的映射关系，也就是专门用一个表去做索引。这是可以的。

你说 “Sharding一般只用于检索主键的场景吧？”，这个也很片面。sharding之后的数据，通常来说，用的最多的是，给定一个“sharding key”的情况下，可以顺便做其他的检索。比如存储 Tweet 的评论，可以按照 Tweet\_id 进行 sharding（这样所有跟某个帖子相关的评论都放在一起），然后你可以指定这样的查询：找到123这个帖子，在某个时间戳 xxx 之后的100条评论

select \* from comments where tweet\_id = 123 and timestamp > xxx limit 100;

所以在这种情况下，你仍然可以对其他的column 进行范围查找之类的。但是前提是，你必须给定一个 sharding key，也就是tweet id的情况下才能做其他column的查询。不能在不指定 sharding key 的情况下进行。

这种例子生活着很多，大部分的数据，其实从用户的角度来说，都至少指定了某个用户的xxx，或者某个帖子的xxx。很少直接去筛选非sharding key的column。

另外sharding key 并不是一定要是主键(primary key)，可以是任何你想要作为sharding依据的东西。

**Design a cache for large object >1MB using memcached.**

要求memcache 的data 不能大于1MB per key.我的思路是把文件划分为一个个1mb大小的块，存在磁盘上？然后再在memcache 中建立一个 key 到块的表， 然后再把块cache 到memcache 中？

存在磁盘上肯定是不对的，让你设计Memcache，结果你用了磁盘，那不慢爆了么。而且题目也说了using memcached，也就表示你只有一个存储结构也就是memcached可以用，没有disk提供给你。

一个简单的方法是，把key加suffix，变成多个key。举个例子：

如果 key="hello" value="data = 2M"，那么把key拆分为 "hello-0" 和 "hello-1"，然后把数据的前 1M 和 后1M 分别存在这两个 key 里。

在具体实现的时候，需要注意一下因为 memcached 有可能会自动delete一些数据，根据他的淘汰策略。所以你在访问一个 > 1M 的数据的时候，需要遍历所有的key，在所有的key都存在的情况下，才返回。但凡有一个key被删掉了，就要认为整个key都被删掉了。

然后还有一个问题就是在get的时候如何知道一共有多少个key，那就可以单独把有多少个key存在这个key不加suffix的value里。也就是说上面这个数据，在memcached里会存成三个部分：

key="hello" value=2

key="hello-0" value="the first 1m data"

key="hello-1" value="the second 1m data"

Cassandra and Consistent hashing:

问题1：在Cassandra中，假设有几个row\_key 都hash 到同一个机器，那么我们怎么知道哪些column key 是属于我们用于查找的那个row\_key 呢？

问题2:在consistent hashing 中，当加上了一台新的机器，随机设定k 个 virtual node之后，系统会在这K个virtual node 旁边的node（顺时针方向） 提取data。 问题是这个提取data 的过程， 是需要对它们的旁边的NODE 进行一个full scan 吗？例子：目前只有一台机器A，然后新加一台机器B， 那么要对A进行FULL SCAN 提取合适DATA 到 B 吗？

问题1:一条数据是由 row\_key + column\_key + value 组成的，row\_key 有 routing的作用，但不代表去了这个机器之后，就会把row\_key 丢掉

问题2：对，Full scan，否则你不知道哪些数据应该留在B哪些应该去 A。

Consistent Hashing，一致性Hash算法，是用于数据分片(sharding)，也就是说，一台机器存不下，需要存到多台机器的时候，按照什么样的算法去分.Consistent Hashing 与数据备份(Replica)没有直接的关系。当我们 scale 一个系统的时候，需要做的事情主要是两件：

数据分片 (Sharding)。解决办法之一就是 Consistent Hashing

数据备份 (Replica)。解决办法之一就是一式三份。

在典型的分布式数据库中，一式三份的replica的方法，是“利用”了consistent hashing这个算法，在环上顺时针取3个点来存，这样比较均匀分布。这是一个利用与被利用的关系。

Master-Slave，这种架构一般来说有两种用处：管理机器和数据备份

在管理机器的时候，参考 GFS 和 BigTable 都有一个 Master Node 用于管理slave Node的健康情况，负责重启，分配任务之类的。

在数据备份的时候，就是比如 MySQL 这种数据库自带一种 MasterSlave 的模式，他可以把某一台数据库，备份个几份，这几份就是 slave，这些slave也可以接受用户 的queries。

自己搞定sharding是大部分 nosql 的基本功能,少部分像memcached和redis这种缓存nosql没有,而mysql之类的话要自己写程序去实现这个sharding. Consistent hashing每台机器负责的virtual node是随机分配.不随机的话，数据不就都被放在一起了么。virtual nodes就是为了让数据均匀的分不开。所以我们课上讲的时候，说的是：在 Consistent Hashing Ring 上随机撒1000个点。

In redis, all data has to be in memory. this is the point which is totally different from other No-SQL.usually when you access and read some data in a database, you don't know if the data is in memory(cache) or not.But the case of Redis, it's guaranteed that all data is in memory.and writing to disk is one of the option you can do. which means data on memory is the trunk and data on a disk is kind of backup, So you might lost data which is saved after last saving to a disk if you suddenly turn off the server.At the same time the two on-disk storage formats (RDB and AOF) don't need to be suitable for random access, so they are compact and always generated in an append-only fashion (Even the AOF log rotation is an append-only operation, since the new version is generated from the copy of data in memory).RDB 持久化可以在指定的时间间隔内生成数据集的时间点快照（point-in-time snapshot）AOF 持久化记录服务器执行的所有写操作命令，并在服务器启动时，通过重新执行这些命令来还原数据集。 AOF 文件中的命令全部以 Redis 协议的格式来保存，新命令会被追加到文件的末尾。 Redis 还可以在后台对 AOF 文件进行重写（rewrite），使得 AOF 文件的体积不会超出保存数据集状态所需的实际大小。Redis 还可以同时使用 AOF 持久化和 RDB 持久化。 在这种情况下， 当 Redis 重启时， 它会优先使用 AOF 文件来还原数据集， 因为 AOF 文件保存的数据集通常比 RDB 文件所保存的数据集更完整

**Design email service like gmail.(Email和online chat差不多啊)**

至少可以答出跟message service 类似的部分吧，有thread table, message table,和user table  
Thread table sql  
 user\_id thread\_id sender\_name first\_50\_chars\_of\_last\_message timestamp  
message nosql  
 row key user\_id   
 column key thread\_id + timestamp  
 value content

When a message is sent as new thread or reply of existing thread, the thread table is updated for all users.  
The message is also inserted to message table for each user.

如何给不同email设定label并且存储. 设计 label 并存储就是一个典型的 ManyToMany 的设计。需要两个table。一个是label的table。另外一个是 EmailHasLabel 的 Table。比如：

email\_id / label\_id

1 / 2

1 / 3

代表 email 1 有 label 2 和 label 3.

如果设计email的时候，让你在client里面加入一个"important" folder应该怎么做?gmail里，所有的标记，所有的folder，都是tag。这样的设计之下，更容易scale。如果在之前的table中加一个叫做 is\_important 的field，这种方法固然也可以，但是如果还有其他的新需求，比如 is\_trash，等等，就又要加 新的 数据库 column，就极为不便

**Design Instagram的图片分享程序**

和Design Twitter几乎一模一样。只不过你需要单独在设计一下如何上传照片这个过程。一般来说可能会绕过这个部分，主要问你如何“分享”，也就是如何 Social（follow，unfollow，news feed, timeline）

设计 instagram offline 看照片功能

Instagram的这个功能，不仅支持缓存图片，还支持了在离线状态和网络状态不佳的情况下，继续点赞和评论，使得下载重新连接的时候，能够发往服务器。我觉得可能有两个考点：

1. 图片很大，我们如何尽可能多的在本地客户端缓存，如何在线的时候尽可能多的pull内容，这里可能需要压缩图片

2. 我们可能刚在对离线的评论和点赞存储了一个timestamp，当用户重新连接的时候，提交到服务器上。

**Uber Design**

Uber使用Google的S2 Geometry Library，S2能够将球体分为小区cell，每个小区有一个id，地球大致是一个球形。S2有两个重要特性：它能够定义每个cell的分辨率，它能发现需要覆盖区域的cell。Uber使用3,31平方公里的cell来分片其数据。所有这些让Uber降低乘客等待时间，司机的额外驾驶以及到达乘客招车点的时间(ETA)，当一个乘客使用Uber会发生什么？Uber会使用乘客的当时地理位置和S2的面积覆盖函数来寻找其周围适配的司机，Uber然后选择更短的ETA， Uber寻找的不仅是可搭载乘客的司机，而且还包括那些正在行驶到目标地可搭顺路车的司机。

Ringpop使用TChannel作为其RPC协议.Uber已经从HTTP+JSON迁移到基于TChannel的Thrift，TChannel在Node.js中比HTTP快20倍. Ringpop能够在应用级别实现调度平台服务的水平分区sharding.这个额外的抽象层框架，通过众多Riak(a distributed NoSQL database)粉丝们熟悉的一致性哈希环维持，这样就提供了一种可以在不会中断服务的前提下额外增加新调度服务的手段。Ringpop是一个可嵌入的服务器，能够可靠地对你的数据进行分区，探测节点失败(类似ZooKeeper 服务动态定位器)，能够发现集群中可用的服务。Ringpop通过实现SWIM gossip 协议来跨节点维护一致性视图，集群中任何改变能够被发现并通过这个协议发布到其他节点上。ringpop也可以提供一个代理，将客户端请求route路由分发到后端微服务上。TChannel是Ringpop的gossip和代理能力的基础传输组件

Consistent Hashing 最开始是用来做存储级别的consistent的。Uber用它来做服务级别的consistent。我感觉也就uber有这个需求。其他的公司不需要服务级别的consistent。Uber希望同一个用户的两次请求去的是同一台服务器，这样可以利用一些之前存在那台机器上的数据来加速。而大部分的网站架构，都是可以去不同的机器。

Uber的服务地域性非常强（除了user system 和payment），比如纽约和旧金山的app server完全没必要互通，各干各的工作就好了，更别说不同国家的server了.代码都是一样的，但是服务（servers）确实可以分开搭建，各自不影响。 Uber确实也是这么做的，分不同的 Region，每个 Region 负责一些 City。他们在实际操作中，会把 City从一个 Region 挪到另外一个 Region，以解决某个 Region 过热的问题。

TChannel <https://github.com/uber/tchannel-java>

TChannel is a networking framing protocol used for general RPC, supporting out-of-order responses at extremely high performance where intermediaries can make a forwarding decision quickly. It is easy to implement in multiple languages, especially JavaScript and Python.

TChannel uses a request/response model with out-of-order responses, where slow requests will not block subsequent faster requests at the head of the line. Large requests/responses may be broken into fragments to be sent progressively. TChannel can transport mutliple protocols between endpoints, such as HTTP+JSON and Thrift.

TChannel is a network protocol with the following goals:

request / response model

multiple requests multiplexed across the same TCP socket

out of order responses

streaming request and responses

all frames checksummed

transport arbitrary payloads

easy to implement in multiple languages

near-redis performance

Ringpop is doing all its gossip over TChannel based **persistent connections**. These same persistent connections are used to fanout or forward application traffic.TChannel is also used to talk between services.

Celery is in Python, it is very much like SpringXD in Java. <http://agiliq.com/blog/2015/07/getting-started-with-celery-and-redis/>

Celery is an asynchronous task queue/job queue based on distributed message passing.It is focused on real-time operation, but supports scheduling as well.The execution units, called tasks, are executed concurrently on a single or more worker servers using multiprocessing. Tasks can execute asynchronously (in the background) or synchronously (wait until ready).

Breaking a large task consisting of several independent parts into smaller tasks. eg: Consider you want to read a user's FB timeline. FB provides different endpoints to get different kind of things. FB provides one endpoint to get pictures on a user's timelines, another endpoint to get posts on a user's timelines, another endpoint to get likes of a user etc. If you write a single function to sequentially hit 5 endpoints provided by FB and if network calls take 2 seconds at an average, then your function will take 10 seconds to complete. So you can split your work in 5 individual tasks(it's very easy to do as we will soon see), and let Celery handle the tasks. Celery can hit these 5 endpoints parallely and you can get the response from all the endpoints within first 2 seconds.

With local storage, web applications can store data locally within the user's browser.Before HTML5, application data had to be stored in cookies, included in every server request. Local storage is more secure, and large amounts of data can be stored locally, without affecting website performance.Unlike cookies, the storage limit is far larger (at least 5MB) and information is never transferred to the server.Local storage is per origin (per domain and protocol). All pages, from one origin, can store and access the same data.

// Store  
localStorage.setItem("lastname", "Smith");  
// Retrieve  
document.getElementById("result").innerHTML = localStorage.getItem("lastname");

**Write-ahead-Log on HBase(An open source implementation of BigTable)**

The WAL is the lifeline that is needed when disaster strikes. Similar to a BIN log in MySQL it records all changes to the data. This is important in case something happens to the primary storage. So if the server crashes it can effectively replay that log to get everything up to where the server should have been just before the crash. It also means that if writing the record to the WAL fails the whole operation must be considered a failure.

High level view of how this is done in HBase. First the client initiates an action that modifies data. This is currently a call to put(Put), delete(Delete) and incrementColumnValue(). Each of these modifications is wrapped into a KeyValue object instance and sent over the wire using RPC calls. The calls are (ideally batched) to the HRegionServer that serves the affected regions. Once it arrives the payload, the said KeyValue, is routed to the HRegion that is responsible for the affected row. The data is written to the WAL and then put into the MemStore of the actual Store that holds the record. And that also pretty much describes the write-path of HBase.写WAL是同步操作，blocking的，只有写Log成功了才会执行下一步

Eventually when the MemStore gets to a certain size or after a specific time the data is asynchronously persisted to the file system. In between that timeframe, data is stored volatile in memory. And if the HRegionServer hosting that memory crashes, the data is lost... but for the existence of what is the topic of this post, the WAL!

**TinyURL 可以按地域进行sharding**

long2short 如何知道数据在哪个服务器里?

long url的样子是：http://www.sina.com.cn/xxxx 这个url你一看就知道是中国的对吧？为什么呢，因为sina是中国的。所以web server上存一张常量表，把哪个网站是哪个国家的，放在哪个服务器，配置一下就好了。你可能会问，这么多网站，难道都需要存在这张表里么（上千万个网站），答案是不用，因为根据long tail theory，80%的内容都是20%的少数网站提供。这里我们甚至只需要存比如 Top 10k 或者 Top 1M 的网站就可以了。不会占据多少内存空间，每次web server启动的时候就导入这张表。而且能覆盖掉99%的请求。

不在这张表里的网站怎么写？

不在这张表里的网站说明不经常访问，那么你可以仍然按照之前的方法，直接对整个longurl 或者 domain name 进行 hash。然后来放在对应的机器上。

不需要存储这些不常访问的网站。

**Google Doc**

基本思想和 git 的原理差不多。每一次的edit，都是一次 commit。在用户 edit 的时候，Lock用户所edit的那一行。其他用户就无法编辑。这是一种比较安全的方法。Google 的话，锁定的部分要更细一些。比如连续的一小段。如果真的出现了冲突，可以提示用户，让用户自己选择用哪个版本。

Google docs 的file并不是直接你前端更改了就和GFS沟通，一定是你前端更改，然后有一个google doc service(包括很多server) 和google doc的server再和 gfs进行沟通。另外chunk只会写入，不会更改，要更改的话是类似于bigtable 一样的一套机制，再写一个更改的记录来做。

侧重点在于如何保证每个人看到并修改的是同一个版本。对于实时性，我觉得是每台机器和服务器进行长连接。对于一致性我现在有两个思路：

1. 类似于Git的工作方式，每个人每次修改都会发送消息至服务器，这个消息包含修改的内容还有版本号。服务器收到来源不同的消息后，把一定时间内（比如0.1s）的修改，根据版本号和消息内容进行merge成一个修改，然后统一发送给所有用户。

2. 我看到网上说Google Docs实际使用一种叫做Operational transformation的方法，大意就是每个人都对就近的服务器上进行直接修改，然后这个修改再从最近的服务器传播到其他服务器，其他服务器收到修改消息后根据某种Operational transformation的方法把修改的参数transform成与本地版本匹配的参数。

**Master-slave 可以有多个master吗?**

1.一般都不用多个master，因为master之间的数据很难保证一致性。

2.实际公司很多用master slave的，因为master slave已经足够好用了。一般情况下的设计都是1个master，多个slave，slave的设计一般也是可读的，分担master读的压力。但是可以设计多个Group，每个Group中是1个master和多个slave。通过扩充Group扩大存储。

3.master挂了以后不会single point failure，因为他有slave。promote slave是很快的。不过要人为操作。如果服务对稳定性要求很高很高的话，就需要牺牲一点一致性，改用 NoSQL DB那种数据库，就不是master slave的模式，稳定性（Availability）会高，但是牺牲了一致性（Consistency）。 Cassandra就是Peer to Peer. 写可以写到任何一个上，读也是随便读

Master-slave数据如何同步.用Redis举例，slave启动以后，建立主从链接，会定期从Master上读数据。而且slave第一次同步master的时候，slave会阻塞。一旦slave发起SYNC（同步）请求后，master会把近期修改数据的所有命令，做一个快照，把快照传给slave，slave得到快照后，在自己的机器上恢复数据。In AWS elasticache, there is replication lag == 0/1

**在数据库中存储层级结构 <http://qinxuye.me/article/storing-hierachical-data-in-database/>**

Organization table

ID, Name, Level, Father,

Staff Table

ID Name, OrganizationID

如果简单来做，可以只记录一个staff所在的最底层的OrganizationID。此时看似检索一个大的organization可能需要点时间，但是复杂度其实并不会很高，因为公司人数最多几万人，而且我们如果还建立了索引的话，会更快。

如果想加速的话，可以记录一个staff的OrganizationLevels，例如1.2.3，这样检索的时候就很快了

**Design Excel:**

这题会分两个侧重点来问，当然get和set是最基本的，

1. 处理各个cell之间的dependency，比如cell(1,3)是用公式算出来的cell(1,3)=cell(0,0)+cell(0,1)+cell(0,2)，我会用两层哈希表表示整个表格(Map<int,Map<int,Cell>> workbook)，然后每个Cell中保存一个Set<Cell> parents；(所有计算当前Cell需要依赖的cells，上例就是cell(0,0)，cell(0,1)和cell(0,2)) 和 Set<Cell> children (所有依赖这个Cell通过公式计算出来的cells)，每次改变cell的值就要对children和parents做相应的改变；

2. 处理add或delete一整行或一整列，我会用2d数组，List<List<Cell>> workbook, add的话就直接append，delete行的话就直接erase对应的行，delete列的话就根据列下标，对每行进行erase，好写，但是效率有点低。对于删除一列，可以把每个cell做成一个四向的指针（指向上、下、左、右的cell），这样删除一列可以成为o(k)复杂度，而不是o(mk)

**Sharding == Horizontal Partitioning**

Partitioning is a general term used to describe the act of breaking up your logical data elements into multiple entities for the purpose of performance, availability, or maintainability.

Sharding is the equivalent of "horizontal partitioning". When you shard a database, you create replica's of the schema, and then divide what data is stored in each shard based on a shard key. For example, I might shard my customer database using CustomerId as a shard key - I'd store ranges 0-10000 in one shard and 10001-20000 in a different shard. When choosing a shard key, the DBA will typically look at data-access patterns and space issues to ensure that they are distributing load and space across shards evenly.

"Vertical partitioning" is the act of splitting up the data stored in one entity into multiple entities - again for space and performance reasons. For example, a customer might only have one billing address, yet I might choose to put the billing address information into a separate table with a CustomerId reference so that I have the flexibility to move that information into a separate database, or different security context, etc.

To summarize - partitioning is a generic term that just means dividing your logical entities into different physical entities for performance, availability, or some other purpose. "Horizontal partitioning", or sharding, is replicating the schema, and then dividing the data based on a shard key. "Vertical partitioning" involves dividing up the schema (and the data goes along for the ride).

**Difference between partition key, composite key and clustering key in Cassandra?**

The primary key is a general concept to indicate one or more columns used to retrieve data from a Table.

The primary key may be SIMPLE

create table stackoverflow (

key text PRIMARY KEY,

data text

);

That means that it is made by a single column.

But the primary key can also be COMPOSITE (aka COMPOUND), generated from more columns.

create table stackoverflow (

key\_part\_one text,

key\_part\_two int,

data text,

PRIMARY KEY(key\_part\_one, key\_part\_two)

);

In a situation of COMPOSITE primary key, the "first part" of the key is called PARTITION KEY (in this example key\_part\_one is the partition key) and the second part of the key is the CLUSTERING KEY (key\_part\_two)

Please note that the both partition and clustering key can be made by more columns

create table stackoverflow (

k\_part\_one text,

k\_part\_two int,

k\_clust\_one text,

k\_clust\_two int,

k\_clust\_three uuid,

data text,

PRIMARY KEY((k\_part\_one,k\_part\_two), k\_clust\_one, k\_clust\_two, k\_clust\_three)

);

The Partition Key is responsible for data distribution across your nodes.

The Clustering Key is responsible for data sorting within the partition.

The Primary Key is equivalent to the Partition Key in a single-field-key table.

The Composite/Compound Key is just a multiple-columns key

Amazon 1 hour delivery system, very similar to Uber.

在讲Uber的课上，我们也提到过，我们很难去想和回答Uber自己这套系统是如何设计的，所以一般情况我们都可以从一般性的角度来分析这个问题。系统班课上讲的Uber的service系统还是值得借鉴的。

从User的角度来讲

登入系统下单，创建Order，用户可以自己选择最近的站点或者系统给用户选择最近的站点（如果站点可以提供）

一旦Order被确认和准备投递，需要一个PubSub服务系统用户。

从Driver的角度来讲，如果他是空闲的，则由Driver等待合适的订单，找到合适的订单，由这个Driver负责 delivery。当然还有的策略就是Order等待，等待出现附近的Driver空闲，然后match。可以开脑洞思考各种优化的策略。

Uber设计的Dispatch Service这里就可以应用，可以用来Cache一些公有信息。

从Storage上来看，我们需要一个Order的Tabble，buyer的Table，Driver的Table

Order的设计大致如下：

Order：

fk buyer

fk Driver （等待Driver匹配上后填写）

repeat field Items

投递规划等等信息

Buyer:

用户的基本信息的Table

Driver:

Driver的基本信息，

bool status 是否空闲

然后对于amazon这样大的公司，数据规模很大，然后可以回答我们如何sharding这些数据，按照city，或者某些区域来sharding数据。如何做就近位置的查找，是否可以使用Geo Hash，如何周围最近的topk个近的站点

Shard Graph

图当中最重要的两个概念是 点 和 边，也就是所谓的对象和对象之间的关系。TAO的核心就是在处理对象和对象之间的关联。所有的对象Node和关联Edge都有一个唯一的ID标识，就是每一条数据的Primary ID。（TAO的底层应该还是MySQL），如何做sharding，实际还是以这些ID为key，根据ID放在不同的shard上。（sharding就是这么做的，然后这么做会遇到问题，得考虑怎么解决）

TAO内距离近的数据中心形成一个Region，每个Region有一个缓存，负责这个Region内的所有数据缓存，这个和SQL的缓存不一样的地方是，之前是一个缓存负责一个DB，这个是一个缓存得负责Region内的所有db。这样也有利于避免频繁的跨数据中心的访问。

TAO的数据存储主要三层，底层是MySQL,数据要分片，放不同的Shard上，上层为Leader Cache，也就是主Cache，负责和DB通信，主Cache的上层分布着Follower Cache 从Cache。一般情况是多个Follower Cache对应一个Leader Cache。这样的多级Cache为了降低Cache的耦合度。

从Cache基本上响应了大部分的读请求，读数据就是先看从Cache，没有就差主Cache，没有直接从DB上query。那么对于写操作，我们是要转发到其他备份的Region上，发一个通知，通知大家更新主Cache，更新DB，然后更新从Cache。这里会出现数据不一致的情况。这里的Cache基本上解决了一分部查询效率的问题。

然后就是你已经提到的，我们有时候会进行朋友朋友的朋友的查询，那么这就会发生多跳查询，这时候在数据库内建立二级索引肯定是远远不够了的，因此Facebook有一个分布式图查询引擎Dragon，做了一个Social Hash (相关论文：https://www.usenix.org/system/files/conference/nsdi16/nsdi16-paper-shalita.pdf) 他可以把用户和所有他的朋友都sharding到同一台机器上面。Social Hash大意就是把社交网络里的节点根据不同的连接性质进行重新分片, 有朋友关系的对象和关联，就能分片到同一台机器上，避免这种多跳查询 multi-hop.

现在rider发出一个请求，然后会在所有覆盖的cells里面找available的driver。dispatch server 有了rider的位置和附近的driver就能通过历史交通信息估算出driver pick up rider的时间了，然后根据ETA进行排序，选择哪一个driver去接rider。

Post trip pipeline. A lot of processing must happen after a trip has completed.

Collect ratings.

Send emails.

Update databases.

Schedule payments.

In airports they have to emulate a virtual taxi queue. Supply must be queued in order to take into account the order in which they arrive.

The new dispatch system has two major services: **supply**, the drivers, and **demand**, the riders. These services track all the capabilities and the state machines of supply and demand. For instance, the supply service knows how many seats a vehicle has or if it can fit a wheelchair. The **dispatch system** has a third service, called Disco (**Dispatch Optimization**), whose main function is to match supply and demand. Disco enables Uber to "look into the future" and to use information as it comes in. For instance, the old dispatch system only looked to current available supply. As most partners are usually busy, this approach allowed Uber to maintain a global index. The new dispatch system is more efficient, but it requires much more data. Uber wants this new system to handle one million writes a second and a much higher read rate, so it needed to shard its data.

To achieve that kind of scale, Uber chose to use Google's S2 Geometry Library. S2 is able to split a sphere into cells, each with an id. The Earth is roughly spherical, so S2 can represent each square centimeter of it with a 64-bit integer. S2 has two important properties for Uber: it is possible to define each cell's resolution and it is possible to find the cells that cover a given area. Uber uses 3,31 km2 cells to shard its data. All this new data enables Uber to reduce wait times, extra driving by partners and the overall estimated times to arrival (ETA). So, what happens when a rider wants to use Uber? Uber uses the rider location and S2's area coverage function to look for drivers that can be matched with a rider. Uber then chooses the shortest ETA, taking into account not only the drivers who are available, but also those that will become available in time to pick up the rider.

The dispatch system is mostly built with NodeJS, meaning that it is single-threaded. Uber wants to take advantage of all cores of a machine, but it also needs to add new nodes to the system with ease. Ranney also argues that servers need to be stateful, or else the datastores won't be able to cope with the load. Uber thus opted to treat all Dispatch processes the same, whether they are running on the same machine or not. They've built ringpop to handle this problem. **Ringpop** uses a consistent hash ring, also used by Amazon's Dynamo, memcached or Riak, to distribute state across nodes. To manage cluster membership and failure detection, ringpop uses SWIM, which stands for Scalable Weakly-consistent, Infection-style Process Group Membership Protocol. It is the same gossip protocol that's used by Hashicorp's Serf. Ringpop uses TChannel, also built by Uber, as its RPC protocol.

TChannel is inspired by Finagle's multiplex RPC protocol, Mux, which was built by Twitter. Uber felt the need to create its own protocol mainly because it needed to support multiple languages (javascript and python), tracing and encapsulation. Ranney told the audience that Uber is moving out of HTTP+JSON and moving towards Thrift over TChannel. Ranney claimed that TChannel is twenty times faster than HTTP when used in NodeJS.

Most of Uber's architectural choices are driven by availability and performance, as it is easy to drivers and riders turn to the competition. At Uber, everything has to be retryable, thus, idempotent and killable, including databases. Each piece of the system must be built on the assumption that the only way to shutdown a process is by crashing. All these constraints also favour small services so that if any one crashes, then the disruption is contained.

The proliferation of small services and the extreme distribution of them can have an impact on performance: the overall latency of a request is greater or equal than the latency of the slowest component. Ranney likes Google's Jeffrey Dean approach on this subject. For instance, TChannel supports "backup requests with cross server- cancellation". This means that the same request might be sent to two instances of the same service, with a slight delay between the two. The first instance to reply handles the cancelling the request on the second instance, to cut redundant work.

Uber's approach to data center failure is ingenious. No data is replicated across data centers, as that puts a lot of constraints on availability and consistency. Uber uses the driver's phones to distribute the data. Given that the driver's phones post location updates to the server every four seconds, the server periodically replies with an encrypted state digest. If a data center fails the driver will contact a new data center to post a location update. The new data center doesn't know anything about this particular driver so it asks for the state digest and picks up from there.

the dispatch system is mostly built with NodeJS, but Ranney mentioned Uber wants to switch to io.js, a NodeJS fork. Ranney also briefly talked about other Uber's architecture components. Maps and ETAs are written in several languages, such as C++ and Java, due to the need to integrate with different kinds of services. All their business logic is written in Python. Uber is building their own column-oriented distributed data store but they also use Postgres, Redis, MySQL and Riak.

Uber ETA: <https://eng.uber.com/engineering-an-efficient-route/>

In Uber’s early days, we used a combination of routing engines (including OSRM) to produce an ETA. (We didn’t have in-app navigation at this point, so we only used it for the ETA and map matching to display vehicle locations.)

We called this service “Goldeta”, which was essentially a model that sat on top of the routing engines and made an adjustment to those original estimates using our own **historical Uber Data of similar routes in time and space**. This solution, which ultimately took into account hundreds of thousands of Uber trips, compared them to the initial routing engine ETA. Goldeta worked better than using any single ETA alone. However, one issue with this approach was the **cold start** problem: when we launch in new cities we didn’t have enough data to inform an ETA offset (for new cities, our ETA used to be less accurate than older cities for precisely this reason).

Goldeta served us well for a time, but as we grew to more cities and services (such as UberRUSH, launched in March 2014, and uberPOOL which we began work on in late spring 2014), it became clear that we needed a dedicated, in-house routing engine built for Uber. So in 2014 we began working on our own all-in-one **routing engine, Gurafu**. Gurafu’s goal? High-performance, highly-accurate ETA calculation specifically for Uber.

Before we launch into specifics, let’s discuss the essence of what you need in a routing engine. The whole road network is modeled as a graph. **Nodes represent intersections, and edges represent road segments**. The edge weights represent a metric of interest: often either the road segment distance or the time take it takes to travel through it. Concepts such as **one-way streets, turn restrictions, turn costs, and speed limits** are modeled in the graph as well.

Design Youtube, 如何设计一个视频分享系统

如何存储

比如Youtube之类的视频网站，存储的都是视频，视频可以说写入以后是不会修改的 immutable，是不可修改数据，且要求很大的吞吐量throughput，对数据一致性要求低 low consistency，一般情况下，我们会选择把视频，照片这些数据存入到了分布式文件系统中 distributed file system。

因为是一个视频分享系统，会有很多对视频的统计数据statistics，这些统计数据往往是real time的，要求实时的，那么我们可以用Nosql数据库在存储这些统计数据。Media Data + Meta Data + Statistics Data

web server设计

这方面我们需要大规模的server的话，可以采用一致性hash(consistent hashing)来做Load balance。在server上，视频的流畅度往往是很重要的，有时候丢包，比如少一些像素，人眼是分辨不出来的，所以我们允许传输的过程中丢包，可以采用UDP协议。

寻找数据流中出现最频繁的k个元素(find top k frequent items in a data stream)。这个问题也称为 Heavy Hitters.

这题也是从实践中提炼而来的，例如搜索引擎的热搜榜，找出访问网站次数最多的前10个IP地址，等等。

Top K：<https://soulmachine.gitbooks.io/system-design/content/cn/bigdata/heavy-hitters.html>

方案1: HashMap + Heap

用一个 HashMap<String, Long>，存放所有元素出现的次数，用一个小根堆，容量为k，存放目前出现过的最频繁的k个元素，

每次从数据流来一个元素，如果在HashMap里已存在，则把对应的计数器增1，如果不存在，则插入，计数器初始化为1

在堆里查找该元素，如果找到，把堆里的计数器也增1，并调整堆；如果没有找到，把这个元素的次数跟堆顶元素比较，如果大于堆丁元素的出现次数，则把堆丁元素替换为该元素，并调整堆

空间复杂度O(n)。HashMap需要存放下所有元素，需要O(n)的空间，堆需要存放k个元素，需要O(k)的空间，跟O(n)相比可以忽略不急，总的时间复杂度是O(n)

时间复杂度O(n)。每次来一个新元素，需要在HashMap里查找一下，需要O(1)的时间；然后要在堆里查找一下，O(k)的时间，有可能需要调堆，又需要O(logk)的时间，总的时间复杂度是O(n(k+logk))，k是常量，所以可以看做是O(n)。

如果元素数量巨大，单机内存存不下，怎么办？ 有两个办法，见方案2和3。

方案2: 多机HashMap + Heap

可以把数据进行分片。假设有8台机器，第1台机器只处理hash(elem)%8==0的元素，第2台机器只处理hash(elem)%8==1的元素，以此类推。

每台机器都有一个HashMap和一个 Heap, 各自独立计算出 top k 的元素

把每台机器的Heap，通过网络汇总到一台机器上，将多个Heap合并成一个Heap，就可以计算出总的 top k 个元素了

方案3: Count-Min Sketch + Heap （Bloom Filter with count）

既然方案1中的HashMap太大，内存装不小，那么可以用Count-Min Sketch算法代替HashMap，

在数据流不断流入的过程中，维护一个标准的Count-Min Sketch 二维数组

维护一个小根堆，容量为k

每次来一个新元素，

将相应的sketch增1

在堆中查找该元素，如果找到，把堆里的计数器也增1，并调整堆；如果没有找到，把这个元素的sketch作为钙元素的频率的近似值，跟堆顶元素比较，如果大于堆丁元素的频率，则把堆丁元素替换为该元素，并调整堆

这个方法的时间复杂度和空间复杂度如下：

空间复杂度O(dm)。m是二维数组的列数，d是二维数组的行数，堆需要O(k)的空间，不过k通常很小，堆的空间可以忽略不计

时间复杂度O(nlogk)。每次来一个新元素，需要在二维数组里查找一下，需要O(1)的时间；然后要在堆里查找一下，O(logk)的时间，有可能需要调堆，又需要O(logk)的时间，总的时间复杂度是O(nlogk)。

方案4: Lossy Counting

Lossy Couting 算法流程：

建立一个HashMap，用于存放每个元素的出现次数

建立一个窗口（窗口的大小由错误率决定，后面具体讨论）

等待数据流不断流进这个窗口，直到窗口满了，开始统计每个元素出现的频率，统计结束后，每个元素的频率减1，然后将出现次数为0的元素从HashMap中删除

返回第2步，不断循环

Lossy Counting 背后朴素的思想是，出现频率高的元素，不太可能减一后变成0，如果某个元素在某个窗口内降到了0，说明它不太可能是高频元素，可以不再跟踪它的计数器了。随着处理的窗口越来越多，HashMap也会不断增长，同时HashMap里的低频元素会被清理出去，这样内存占用会保持在一个很低的水平。

很显然，Lossy Counting 算法是个近似算法，但它的错误率是可以在数学上证明它的边界的。假设要求错误率不大于ε，那么窗口大小为1/ε，对于长度为N的流，有N／（1/ε）＝εN 个窗口，由于每个窗口结束时减一了，那么频率最多被少计数了窗口个数εN。

该算法只需要一遍扫描，所以时间复杂度是O(n)。

Uber的最近常考题，乍看跟Design Uber没啥区别， 只是从Deliver Rider换成了Package. 核心功能还是 1. User request service(myLocation, Restaurant, Order). 2. Driver pick up order and serve the order

这边有几个difference: 1. User request service的时候需要提供Restaurant信息和Order信息。 2. How to select nearest driver? 这边还得考虑Food啥时候ready，因为restaurant准备食物还得一些时间，所以需要Restaurant trigger一个event, Food is ready, 请求Drivers来pick up吧，这时候开始match. 否则Drivers需要在Restaurant等，体验不好。并且Restaurant可以自己派Driver去送货，例如纽约曼哈顿经常见小哥骑车送餐。3. 比起Uber， 这边加了another entity, Restaurant. 所以需要Register Restaurant, Upload Menu. 这个可以提供一个Web Portal给商家。

跟传统Uber一样的是，Payment service, update driver's location every 4 seconds once the package is on the way, Post trip service, like rating, tips.

Client向Server传文件：

1. 一个client 对一个server

2. 多个client 对一个server

3. 一个client 对多个server

4. 多个client 对多个server

可以把1和2算做一类问题，3和4算做第二类问题

第一类问题就是传呗，一个server最终也就那么多cpu/ram/hdd，没啥花样

第二类问题，多个server可以选其中1-2个做master，其余的全是chunk server。

client上传的时候，不要经过master，master直接分配几个chunk server给client传文

件，其中一个chunk server可以作为leader，文件partition之后，一式三份保存在不

同的chunk server；文件分隔和保存记录可以存在master上