* Example:
  + Design Uber
  + Facebook Nearby
  + Yelp
  + Pokemon Go

Please design Uber:

* 是否看过Uber的视频，文章 如何实现的。
  + RingPop: github.com/uber/ringpop-node 相比于Master/Slave, 这是一个去中心化架构
    - 看Short URL
  + TChannel: github.com/uber/tchannel
    - 一个高效的RPC协议： 机器A的某程序去call机器B的程序。跨机器的程序方法调用。 （比HTTP传输更快，Facebook有一个Thrift， Google？？）
  + Google S2：
    - 地理位置信息存储于查询的算法。
    - 如何存储谁在什么地方，以及谁周围有什么（哪些车，哪些人，哪些餐馆，哪些小精灵）
  + Riak:
    - Dynamo DB的开源实现

Read more on **eng.uber.com Uber自己的Engineering 博客**

**RPC：** method A call method B from another machine。 Uber的TChannel / Google Thrift 这些 公司都自己造轮子 来进行server 跨 机器通信

Google S2 一个 地理位置信息存储与查询的 算法。。看Github资料

最重要的看 自己熟悉什么， 而并非是该面试公司的具体实现方法 （至少能work）

数据库熟悉哪个 就说哪个。

System Design = Logic Design（Scenario, sERVICE, Scenario， 一般的backend engineer） + Infrastructure Design （for infrastructure engineer）

用4S法来设计Uber：

* Scenario: 需要设计哪些功能，设计的多牛
  + Features -- 定义 feature （user story）：
    - 第一阶段：(哪些功能必不可分，最主要的 一定要有，login 和 支付 是 common feature，不是独特的）
      * **Driver reports locations**
      * **Rider Request Uber， Match a driver with rider**
    - 第二阶段 -- 更细节的feature：
      * Driver deny / accept a request
      * driver cancel a matched request
      * Rider cancel a request
      * Driver picks up a rider / start a trip
      * Driver drop off a rider / end a trip
    - 第三阶段：
      * Uber Pool （share ride）
      * Uber Eat
  + QPS：
    - 根据有多少辆车，多少城市 来 推断一共有多少QPS
      * Uber 2011 年 的 QPS：uber 刚刚成立1年，只在旧金山范围内有车，4000辆车， 最开始定义的两个 feature 是 最主要的，
      * 2011年：Given 4000 cars，average QPS: 1000 ( 假设司机每隔4秒Driver发一次request到Server，所以 就是 4000 / 4 = 1000 )。乘客需不需要汇报自己的位置，从系统角度，不应该获取 用户的 位置。但是google会偷偷记录你的地理位置。
      * 2015年，假设200K Driver，QPS：200K / 4 = 50K (2015年有1Million user， 同时在线的假设有20%， 所以20万driver同时在线）
      * Peak QPS： 50K \* 3 = 150K
      * 这个QPS大部分是写操作（相比于Tiny URL大部分是读操作）。
    - 存储估值： 存储量计算
* Service：
  + Uber 主要干的事就是两件
    - 记录车的位置 GeoService： geography service 具体几秒，是有trade off的，看到 uber 地图上还有 几百米，车已经到了。因为 有 4s 延迟
    - 匹配打车请求 Dispatch Service：用户Request car
    - Geography Service 和 Dispatch Service 漏了点什么：
      * 需要给 司机 返回matched rider ，但是这样的话 Geography Service 又要给 Driver return 信息，不太符合这个service的设计初衷
    - 可以变化成，不管司机和乘客都直接访问Dispatch Service， Geo Service隐藏在 Dispatch Service 的 后面， 变成Internal Service，不直接暴露给外部web server。 否则GeoService and DispatchService功能会有重复。
* Storage：
  + DispatchService包含 Dispatch Web Server， 去TripTable里面去找 存储的 是一个个 trip。
  + GeoService 包含 Geo Web Server 和 LocationTable
  + 数据表单：
    - Trip ： TripTable的Schema（写比较少，读比较多（司机每4s顺便获得被匹配的trip信息，如果没有实现push notification的话））: trip table 只有 match driver 的 时候，status 改变的时候，才会修改 trip table，所以这个 table 是 读多。 每4s 读一次 trip table 返回 driver 当前 的 trip match 信息（可以是 push notification）。rider 则是 在 request 的时候 返回 一个 matched driver，查询的会 少一点。
      * TripID
      * Driver Id, rider Id
      * Location
      * Status
      * createdAt
    - Location table，每4s driver 地理位置信息 更新 这个 table。如果 10分钟都没有更新，说明 司机 掉线了
      * Driver Key
      * Latitude
      * Longitude
      * updated At(如果很久都没更新，说明司机掉线了)
  + LBS的难点： 如何存储和查询地理位置信息？
    - Naive: 如果按照现在的方式，就是
      * Select \* FROM Location

WHERE lat < myLat + Delta

AND lat ? myLat - delta

AND long < AND long > something

这样查询的话 非常慢： 因 lat 可以存成index， 但是对于 longitude 只能for循环，，基本等于扫描了一遍数据库（需要自己理解 SQL Range Query的实现）

* Google S2:
  1. The Hilbert Curve: 从左下角到右下角Mapping： 在这个图里面，数字越接近的，地图的空间位置也比较接近。将地址空间映射到2^64的整数 （概率上来说比较接近，不需要100%正确，只需要cover 90%的case）。把2维上的点，映射到一维的数字，范围 0 - 2^64. 本来要对 两个 column进行 range query，现在只需要做一个 range query
* Geography Hash：
  1. Peano Curve：和 S2的区别是，S2用 整数代表经纬度，Peano Curve使用字符串来代表经纬度 BASE32:0-9, a-z 去掉a，i，l，o （因为可能会搞错。**公共前缀越长，两个点越接近**。
  2. 为什么用base32： 因为可以用5个2进制位便是
  3. 有一张固定的图，上面用BASE32 encode，你所在的位置首先globally, 分成 32个 格子，左边16个，右边16个，按照 peano curve 的 特点 增加数字，然后zoom in，再分成32个 格子，经度继续提高。google HQ 就是 9q9hvu7wbq2s
  4. Google HQ 和 LinkedIn HQ 前缀共同的有4位。Google 和 FB 前缀共同的有3位。 到第五位 就是 2KM 的距离
  5. 格子 分割 理念
  6. 哪个效果更好： Google S2 Hilbert 效果更好，但是概率上 Peano curve也是 work 的
  7. 实现方法：二分（先经度longitude后纬latitude，经纬交替）。为什么先经度，后纬度，因为经度范围大（-180,180），维度（-90， 90）。切出来的格子越方，里面的距离就会越近，所以先竖着切一刀，再横着切一刀，再竖着切。 经度：靠左0，靠右1。维度靠下0，靠上1。经纬的hashing逐步逼近。凑足5个二进制位，就能map到geoHashing的1个value了，一步步逐步计算，就能得到最终的hashing value
     + 如果 坐标是 （36,27），则 36 在 经度（-180,180）右边，所以是 1， 27 在 纬度 -90，90 的 右边，所以也是 1， 这样做 5次，就相当于 划了 32次，得到了第一位 的 字符， 一共做5次这样的计算，就得到了5个 character
  8. 划5次 32个小格，对应的经度。Wiki有个表格，5个 character 的 距离误差就是 2KM，表示geohash 5个如果一只的话，就表明了相应的距离。 可以将prefix=5作为标准，哪些车辆的geohashing前5位和当前的相等 -- 如何实现？ 内存中实现 的 话 Trie 最好。 但是现在要存在数据库中，Trie就不合适叻
     + 在Google 2KM 范围内打车，就 相当于 在 GOOGLE HQ 画格子， 如果 prefix = 5，说明 经度 2.4 KM。
  9. 如果在内存中，最好的方式就是 Trie了， 但是 现在要在 SQL 中query
  10. 如果用SQL数据库实现的话，
      + 就要在location table加一个司机的GeoHashing 的 Column，并 index。 下面的Query虽然还是有点慢，但是比Naive 要好 SELECT \* from location where geohash LIKE 09hji%. LIKE 的 本质 也是 range query。
      + SQL 另一个问题是driver位置不断更新，B+数不停变动，效率低。并且应避免LIKE query。LIKE Query 挺慢的。建立index的东西不要总去变
  11. 如果用NOSQL - Cassandra:
      + 将geohash 设为column key。不能用 row key，因为 row key 虽然支持index，但是 必须具体指导 row key 是 什么。column key 支持 range query。
      + 使用 range query （9q9hv0, 9q9hvz）
      + 同样的问题： column key是有index的，被index的column不适合被经常修改
  12. 如果用NoSQL - Redis / Memcached（x） 这类key value pair的DB
      + 这里Redis 是 最好的 存储结构：
      + memcached的问题是无法 persistence
      + Driver 的位置存储， 如果是9q9hvt， 则存储在9q9hvt，9q9hv， 9q9h这三个key。 存三份。value是一个 set
      + Memcached会有consistency的问题
      + 用Redis可以实现Set的原子操作。读写速度接近内存访问速度 > 100K QPS。支持persistence
      + redis需要用 两个table， location table 存储 driver 根据 location的inverted index。 driver table 存储每个driver具体的信息。
      + location table 存储3个经度范围的 key。value存储的 是 driver id。 Key: geohash, value: {driver\_id, driver2\_id, driver3\_id}
        - （lat, long） ->
          * geohash: 先找6位prefix 的 entry（1公里范围） -> [driver 1, driver 2, driver3, ...]
          * 再找 5位prefix 的 entry，扩大搜索范围
          * 最后找 4位 prefix entry
          * 最后实在找不到，说 没有人愿意接
      + Driver table: driver key: driver value(properties， trip id). 存3个geohashing。根据 location table query返回 的 driver id 找 driver 的 具体信息
      + Working Solution：
        - 乘客发出打车请求，server create new trip，返回 trip id 给用户，每隔几秒 query 是否请求成功
        - server find matched driver （根据 peano curve 和 redis中存储的 location的 inverted index），写入 trip table中，status改变为 等待司机恢复，同时修改 driver table中 司机状态为 不可用，并在 driver table中 存入 trip id
        - 司机汇报自己位置，顺便再 driver table中 发现有分配给自己的 trip，去 trip table 查询 对应 trip，返回trip 信息给 driver
        - 司机接受打车请求：修改 driver table和 trip table中状态信息
        - 乘客发现匹配成功，获得司机信息
        - 司机拒绝打车请求，server 修改 driver table， trip table中状态信息，标记 该 司机 拒绝了 请求，同时 重新 匹配一个 司机
      + 我们把不长改的 Trip Table 放到 SQL中， 常改的 driver table 和 location table 放到 Redis 中
* Scale:
* 所以： Dispatch Service 有 UserTable， TripTable， Geo Service 有 Location Table and Driver Table
* 有什么隐患： 需求是150K QPS， Redis读写效率 大于 100K QPS，是不是1-2台就好了？并不是
* 如果机器挂了，打车请求就无法满足，损失好多钱。怎么办？
  1. DB Sharding： Avoid single point failure and 分摊流量
  2. 按照什么sharding？
     + - 本来以为 按照 geohash 进行 sharding
       - Uber 按照 城市进行sharding。 为什么不按照GeoHash进行 sharding ？ Uber的服务是基于城市的，主要解决市内交通，解决城市打车问题。跨城市打车 都是 corner case。没有什么request在 天津打北京的车。
       - 如何定义城市？Geography Fence的概念。把城市理解为多边形，判断一个点在不在一个多边形之内 就是 一个计算 几何的 问题。给定一个 loongitude，latitude，判断是不是 在 这个 城市之内。城市数目：400个。每个城市税收，法律也不一样，这也是一个sharding的原因。查找的话，只要for循环就行，for循环400个点
       - **如果乘客站在两个城市的边界怎么办？**找到乘客周围的2-3个城市。这种情况下司机记录在哪个城市关系不大。
* How to check if rider is in Airport? Airport 问题特殊解决
  + - uber 在机场的界面不太一样。如何找到在 机场 的 信息
    - 用GeoFence来代表Airport
    - Airport有上万个，如果直接for循环，会比较慢。先找到城市，再找到Airport。分两个级别找。
* How to reduce impact on db crash?
  1. Uber 最终放弃了Redis 因为 redis 容易挂，用了比较稳定的Riak
  2. 方法1： Replica by Redis
     + Master - Slave replica
  3. 方法2： Replicaby yourself
     + 每份数据写3份
     + sharding key 从 123 扩展为 123-0， 123-1， 123-2
     + 三份数据存在3个不同机器上
  4. 方法3，最好的方法就是 多用几台机器：
     + 1000 台 Cassandra 机器，每个 机器 100 QPS
     + 让更强大的NoSQL数据库（Cassandra， Riak）进行 replica
* Takeaway:
  1. Uber 是一个写密集的应用，与大部分的读密集应用不一样
  2. 解决LBS类的核心问题： Geohash / Google S2
     + 如何存储司机位置
     + 如何查询乘客周围的车
  3. 分析整个打车流程
  4. 按照城市进行sharding
  5. GeoFence 查询
  6. DB crash
  7. Ring Pop
* Redis 特点：
  1. Key, Value两级结构，很快， value支持List， Set。
  2. 经常做Consumer/ Producer 队列。
  3. 最近Redis支持了sharding。
* Cassandra：
  1. rowkey: city-Id， column key(geoHash，支持range query)
  2. Cassandra Notes:
     + Primarikey = (partition key, clustering key) where partition key is used to distribute entries into different nodes, and clustering keys is used for query.

<https://www.datastax.com/dev/blog/a-deep-look-to-the-cql-where-clause>

* Yelp与Uber的区别：
  1. 没有那么多 写 操作。因为 餐馆不会动。读密集，不需要更新地址那么频繁。 （饭店不会动）。
  2. 可以优化的方式： Cache （memcacheD）因为 不变的东西，可以缓存
* DB服务器 存储在 离 城市比较近 的 话，就会 访问较快
* Cassandra DB:
  1. Row key for sharding, column key (range query), value (需要serialize)
* Redis 就是 row, value PAIR。 比其他NOSQL快很多，有100K QPS
  1. redis 支持了 sharding （cluster）， 因为已经sharding 帮你做好了。也可以 用 city id + geohash 做为 key，city id 进行 sharding
* Redis 经常做 Message Queue （producer / CONSUMER problem） ，会 比较快
* Uber 是 ringpop 结构，所以希望 DB 和 server都在 本地