

数据库与聊天系统

Database & Design WhatsApp

课程版本 : v4.1 本节主讲人 : 东邪

九章的课程不允许录像, 否则将追究法律责任和经济赔偿



扫描二维码关注微信/微博
获取最新面试题及权威解答

微信: [ninechapter](#)

微博: <http://www.weibo.com/ninechapter>

知乎: <http://zhuanlan.zhihu.com/jiuzhang>

官网: <http://www.jiuzhang.com>

- Design WhatsApp – User System
 - SQL vs NoSQL
 - Consistent Hashing
- Design WhatsApp – Real-time Service
 - 聊天系统的精髓
- Design WhatsApp – Online Status
 - Pull vs Push
- 这节课之后您可以学会
 - SQL vs NoSQL
 - Consistent Hashing
- 相关设计题
 - Design Facebook Messenger
 - Design WeChat



Interviewer: Design WhatsApp

设计 “在干嘛” 聊天APP



- 基本功能：
 - 用户登录注册
 - 通讯录
 - 两个用户互相发消息
 - 群聊
 - 用户在线状态
- 其他功能：
 - 历史消息
 - 多机登陆 Multi Devices

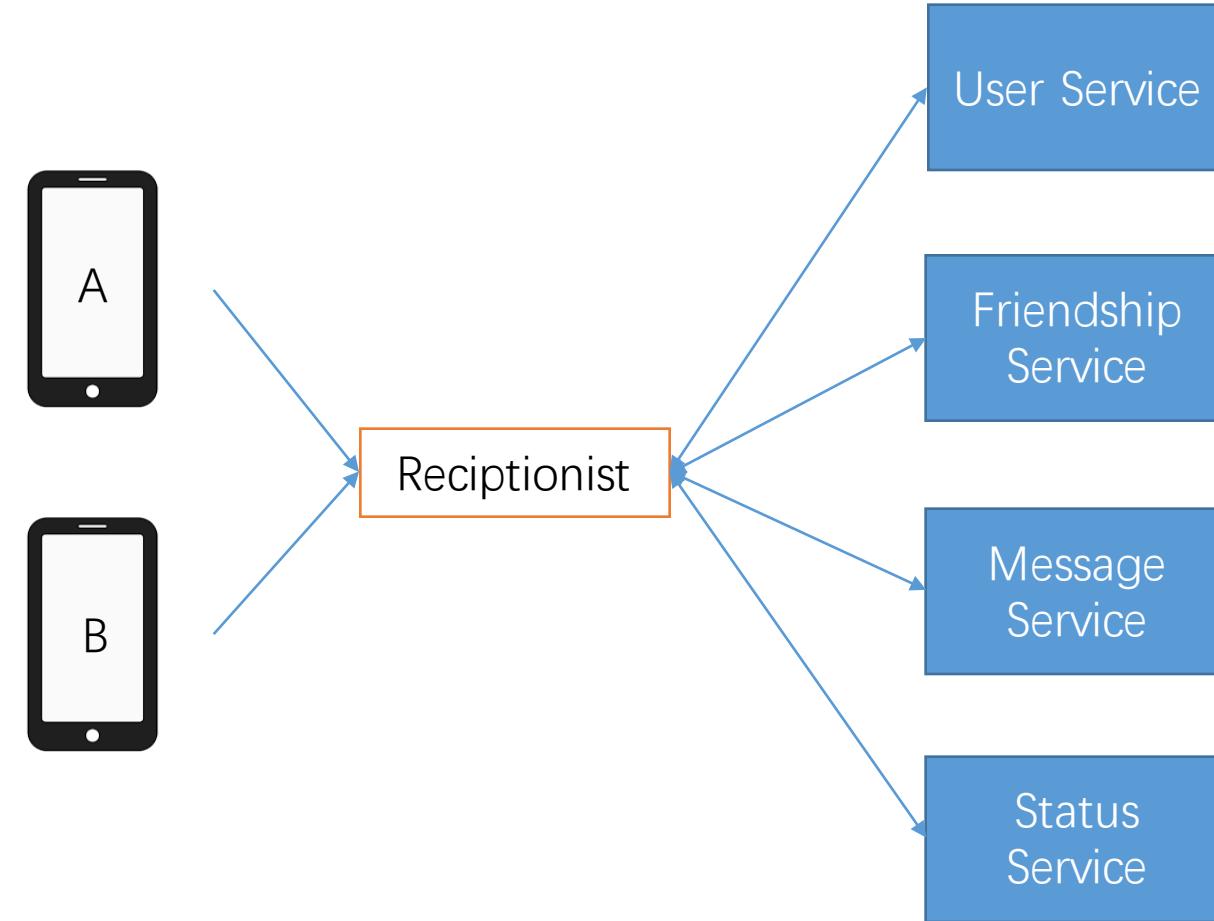


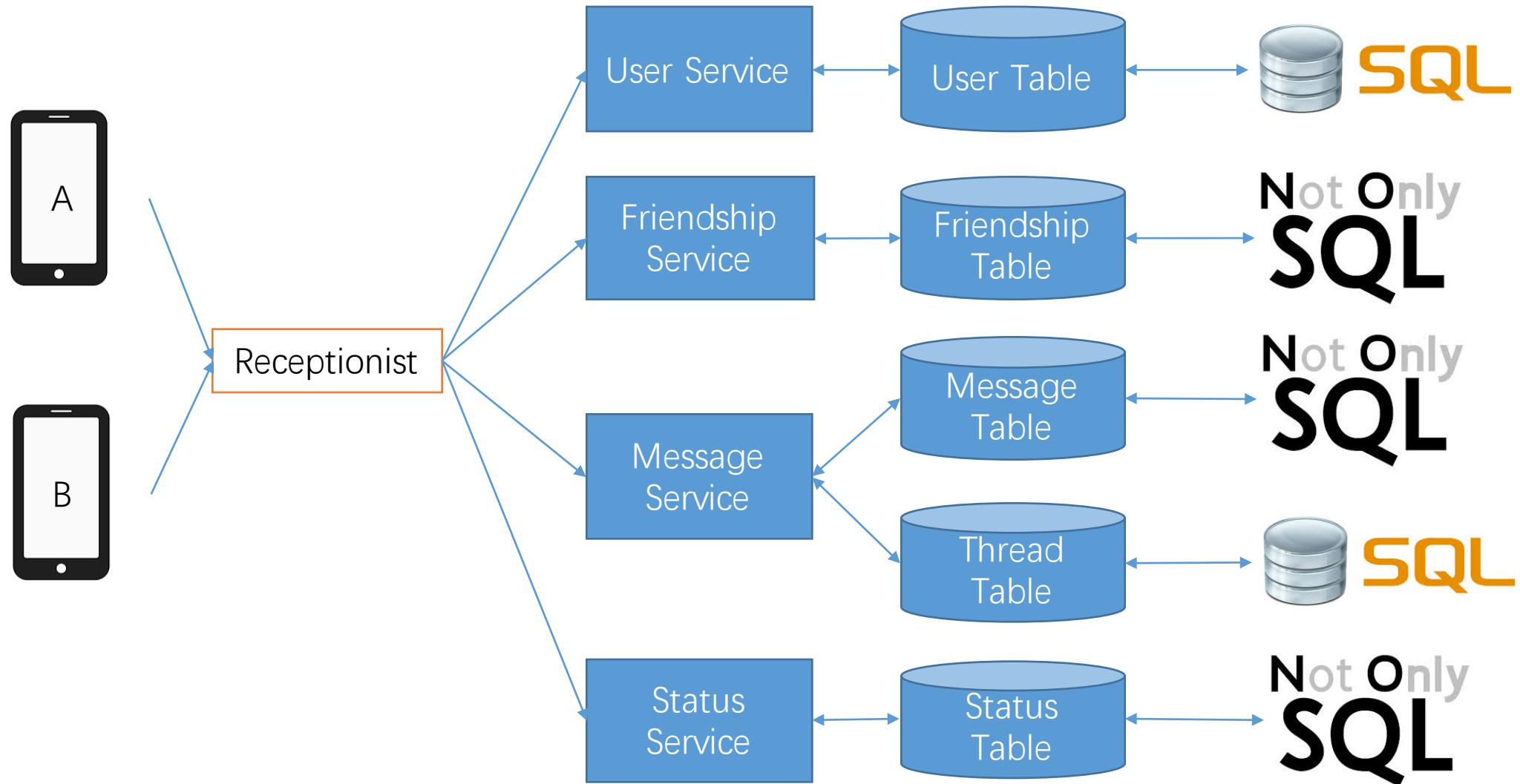
Needs 设计多牛的系统？

- WhatsApp
 - 1B 月活跃用户
 - 75% 日活跃 / 月活跃
 - 约750M日活跃用户
 - ——数据来自 Facebook 官方，截止2016年3月
- 为了计算方便起见，我们来设计一个100M日活跃的WhatsApp
- QPS：
 - 假设平均一个用户每天发20条信息
 - $\text{Average QPS} = 100M * 20 / 86400 \sim 20k$
 - $\text{Peak QPS} = 20k * 5 = 100k$
- 存储：
 - 假设平均一个用户每天发10条信息
 - 一天需要发 1B，每条记录约30bytes的话，大概需要30G的存储

Applications 有哪些独立的服务模块

- User Service
 - 负责登陆注册
- Friendship Service
 - 负责通讯录
- Message Service
 - 负责聊天
- Status Service
 - 负责用户在线状态





是不是有点晕？

通常在系统设计的面试中，不会让你设计一整个完整的系统
而是挑出其中的一块来让你设计

- Design User System 设计用户系统
 - 包括用户的注册、登录
 - 用户数据的存储与查询
 - 好友关系的存储与查询
- Design Message System 设计聊天系统
 - 收发消息
 - 消息数据的存储与查询
- Design Online Status System 设计在线状态系统
 - 用户可以实时看到自己好友的在线状态

Design User System

设计用户系统

实现功能包括注册、登录、用户信息查询

- Scenario 场景
 - 注册、登录、查询、用户信息修改
 - 哪个需求量最大？
- Needs 需求
 - 支持 100M DAU
 - 注册，登录，信息修改 QPS 约
 - $100M * 0.1 / 86400 \sim 100$
 - 0.1 = 平均每个用户每天登录+注册+信息修改
 - Peak = $100 * 3 = 300$
 - 查询的QPS 约
 - $100 M * 100 / 86400 \sim 100k$
 - 100 = 平均每个用户每天与查询用户信息相关的操作次数（查看好友，发信息，更新消息主页）
 - Peak = $100k * 3 = 300 k$
- Application 应用
 - 一个 AuthService 负责登录注册
 - 一个 UserService 负责用户信息存储与查询

QPS 与 系统设计的关系

为什么要分析 QPS?

QPS 的大小决定了数据存储系统的选择

- MySQL / PosgreSQL 等 SQL 数据库的性能
 - 约 1k QPS 这个级别
- MongoDB / Cassandra 等 硬盘型NoSQL 数据库的性能
 - 约 10k QPS 这个级别
- Memcached / Redis 等 内存型NoSQL 数据库的性能
 - 100k ~ 1m QPS 这个级别
- 以上数据根据机器性能和硬盘数量及硬盘读写速度会有区别
- 思考：
 - 注册，登录，信息修改，300 QPS，适合什么数据存储系统？
 - 用户信息查询适合什么数据存储系统？

用户系统特点

读非常多，写非常少

一个读多写少的系统，一定要使用 Cache 进行优化

- Cache 是什么 ?
 - 缓存, 把之后可能要查询的东西先存一下
 - 下次要的时候, 直接从这里拿, 无需重新计算和存取数据库等
 - 可以理解为一个 Java 中的 HashMap
 - key-value 的结构
- 有哪些常用的 Cache 系统/软件 ?
 - Memcached (不支持数据持久化)
 - Redis (支持数据持久化)
- Cache 一定是存在内存中么 ?
 - 不是
 - Cache 这个概念, 并没有指定存在什么样的存储介质中
 - File System 也可以做 Cache
 - CPU 也有 Cache
- Cache 一定指 Server Cache 么 ?
 - 不是, Frontend / Client / Browser 也可能有客户端的 Cache

Mem-Cache

存在内存中的Cache

Memcached

一款负责帮你Cache在内存里的软件
非常广泛使用的数据存储系统

Memcached 使用例子

```
1 cache.set("this is a key", "this is a value")
2 cache.get("this is a key")
3 >> "this is a value"
4
5 cache.set("foo", 1, ttl=60)
6 cache.get("foo")
7 >> 1
8
9 # wait for 60 seconds
10 cache.get("foo")
11 >> null
12
13 cache.set("bar", "2")
14 cache.get("bar")
15 >> "2"
16
17 # for some reason like out of memory
18 # "bar" may be evicted by cache
19 cache.get("bar")
20 >> null
```

Memcached 如何优化 DB 的查询

```
1 class UserService:  
2  
3     def getUser(self, user_id):  
4         key = "user::%s" % user_id  
5         user = cache.get(key)  
6         if user:  
7             return user  
8         user = database.get(user_id)  
9         cache.set(key, user)  
10        return user  
11  
12    def setUser(self, user):  
13        key = "user::%s" % user.id  
14        cache.delete(key)  
15        database.set(user)  
16
```

- 用户是如何实现登陆与保持登陆的？

- 会话表 Session

- 用户 Login 以后

- 创建一个 session 对象
 - 并把 session_key 作为 cookie 值返回给浏览器
 - 浏览器将该值记录在浏览器的 cookie 中
 - 用户每次向服务器发送的访问，都会自动带上该网站所有的 cookie
 - 此时服务器检测到cookie中的session_key是有效的，就认为用户登陆了

- 用户 Logout 之后

- 从 session table 里删除对应数据

- 问题：Session Table 存在哪儿？

- A: 数据库
 - B: 缓存
 - C: 都可以

Session Table			
session_key	string	一个 hash 值，全局唯一，无规律	
user_id	Foreign key	指向 User Table	
expire_at	timestamp	什么时候过期	

Session Table 存在哪儿？

一般来说，都可以，即便存在 Cache 里
断电了相当于让所有用户都 logout 也没啥大不了

存在数据库里肯定更好
如果访问多的话，就用 Cache 做优化即可

- 对于 User System 而言
 - 写很少
 - 读很多
- 写操作很少，意味着
 - 从QPS的角度来说，一台 MySQL 就可以搞定了
- 读操作很多，意味着
 - 可以使用 Memcached 进行读操作优化
- 进一步的问题，如果读写操作都很多，怎么办？
 - 方法一：使用更多的数据库服务器分摊流量
 - * 方法二：使用像 Redis 这样的读写操作都很快的 Cache-through 型 Database
 - * Memcached 是一个 Cache-aside 型的 Database，Client 需要自己负责管理 Cache-miss 时数据的 loading

- 使用 Cache 优化数据库的读操作
- Session (存在服务器端) / Cookie (存在浏览器端)

- 单向好友关系 (Twitter、Instagram、微博)

Friendship Table

from_user_id	Foreign key	用户主体
to_user_id	Foreign key	被关注的人

- 双向好友关系 (WhatsApp、Facebook 、 微信)
 - 方案1：存为两条信息，A关注了B，B关注了A
 - 方案2：存为一条信息，但查询的时候需要查两次
- 好友关系所涉及的操作非常简单，基本都是 key-value：
 - 求某个 user 的所有关注对象
 - 求某个 user 的所有粉丝
 - A 关注 B → 插入一条数据
 - A 取关 B → 删除一条数据

SQL vs NoSQL

Friendship Table适合什么数据库？

SQL 和 NoSQL 的选择标准是什么？

数据库选择原则1

大部分的情况，用SQL也好，用NoSQL也好，都是可以的

数据库选择原则2

需要支持 Transaction 的话不能选 NoSQL

数据库选择原则3

你想不想偷懒很大程度决定了选什么数据库

SQL更成熟帮你做了很多事儿

NoSQL很多事儿都要亲力亲为(Serialization, Multi Indexes, Sharding)

数据库选择原则4

如果想省点服务器获得更高的性能，NoSQL就更好
硬盘型的NoSQL比SQL一般都要快10倍以上

- 如果存在SQL

Friendship Table		
smaller_user_id	Foreign key	双向好友关系中id小一点的,index=true
bigger_user_id	Foreign key	双向好友关系中id大一点的,index=true

- 查询好友关系时
 - 对于给定的 user_id
 - `select bigger_user_id from friendship where smaller_user_id = $user_id`
 - `select smaller_user_id from friendship where bigger_user_id = $user_id`
- 如果存在 NoSQL
- 很多 NoSQL 一般来说不支持 Multi Indexes
- 所以需要拆分为两条数据
 - A的好友有B: `key=A, value=B`
 - B的好友有A: `key=B, value=A`

- Cassandra 是一个三层结构的 NoSQL 数据库
 - <http://www.lintcode.com/problem/mini-cassandra/>
- 第一层：row_key
 - 又称为 hash_key
 - 也就是我们传统所说的 key-value 中的那个key
 - 任何的查询都需要带上这个key, 无法进行range query
 - 最常用的raw_key: user_id
- 第二层：column_key
 - 是排序的, 可以进行range query
 - 可以是复合值, 比如是一个 timestamp + user_id 的组合
- 第三层：value
 - 一般来说是 String
 - 如果你需要存很多的信息的话, 你可以自己做 Serialization
 - 什么是 Serialization: 把一个 object / hash 序列化为一个 string, 比如把一棵二叉树序列化
 - <http://www.lintcode.com/problem/binary-tree-serialization/>

SQL vs NoSQL

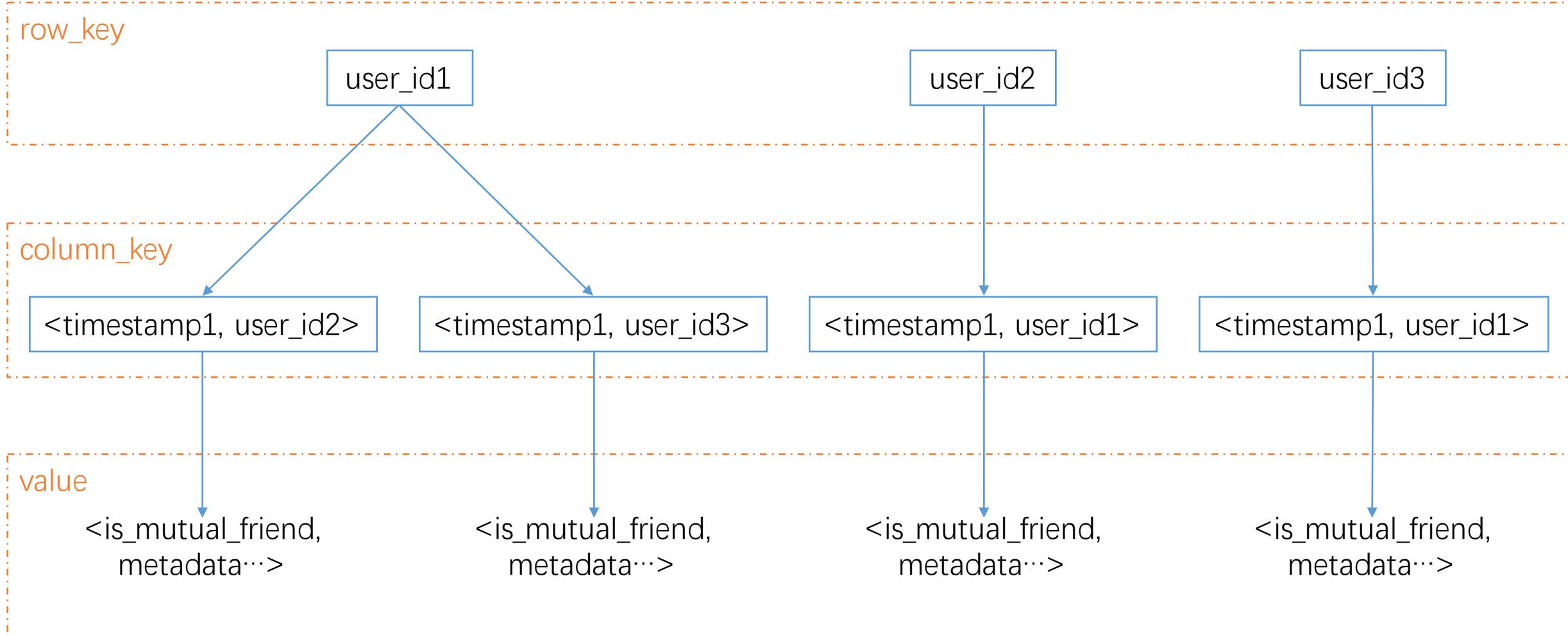
- SQL的column是在Schema中预先指定好的，不能随意添加
- 一条数据一般以 row 为单位（取出整个row作为一条数据）

SQL	id	username	email	password	first_name	...
row1						
row2						
...						

- NoSQL的column是动态的，无限大，可以随意添加
- 一条数据一般以 column 为单位（取出一个column作为一条数据）
- 只需要提前定义好 column_key 本身的格式（是一个 int 还是一个 int+string）

NoSQL	row_key1	column_key1	column_key2	column_key3	column_key4	...
row1						
row2						
...						

以 Cassandra 为例看看 Friendship Table 如何存储



- 使用 Cache 优化数据库的读操作
- Session (存在服务器端) / Cookie (存在浏览器端)
- SQL vs NoSQL的一些基本原则
- Friendship 在 Cassandra (NoSQL) 中如何存储

Interviewer: How to scale?

系统设计的日经题

除了QPS，还有什么需要考虑的？

100M 的用户存在一台 MySQL 数据库里也存得下，Storage没问题

通过 Cache 优化读操作后，只有 300QPS 的写，QPS也没问题

还有什么问题？

单点失效 Single Point Failure

万一这一台数据库挂了
短暂的挂：网站就不可用了
彻底的挂：数据就全丢了

所以你需要做两件事情

1. Sharding
2. Replica

- 数据拆分 Sharding
 - 按照一定的规则，将数据拆分成不同的部分，保存在不同的机器上
 - 这样就算挂也不会导致网站 100% 不可用
- 数据备份 Replica
 - 通常的做法是一式三份（重要的事情“写”三遍）
 - Replica 同时还能分摊读请求

Sharding & Replica in SQL

下面我们以 SQL 型数据库为例
讲一讲 SQL 型数据库是如何做 Sharding 和 Replica 的

数据拆分 Sharding

Vertical Sharding

Horizontal Sharding

纵向切分 Vertical Sharding

User Table 放一台数据库

Friendship Table 放一台数据库

Message Table 放一台数据库

...

稍微复杂一点的 Vertical Sharding

- 比如你的 User Table 里有如下信息
 - Email
 - Username
 - Password
 - push_preference
 - avatar
- 我们知道 email / username / password 不会经常变动
- 而 push_preference, avatar 相对来说变动频率更高
- 可以把他们拆分为两个表 User Table 和 User Profile Table
 - 然后再分别放在两台机器上
 - 这样如果 UserProfile Table 挂了，就不影响 User 正常的登陆
- 提问：Vertical Sharding 的缺点是什么？他不能解决什么问题？

横向切分 Horizontal Sharding

核心部分！

Scale 的核心考点！

一个粗暴的想法

假如我们来拆分 Friendship Table

我们有10台数据库的机器

于是想到按照 `from_user_id % 10` 进行拆分

这样做的是啥？

假如10台机器不够用了

我现在新买了1台机器

原来的%10，就变成了%11

几乎所有的数据都要进行位置大迁移

过多的数据迁移会造成的问题

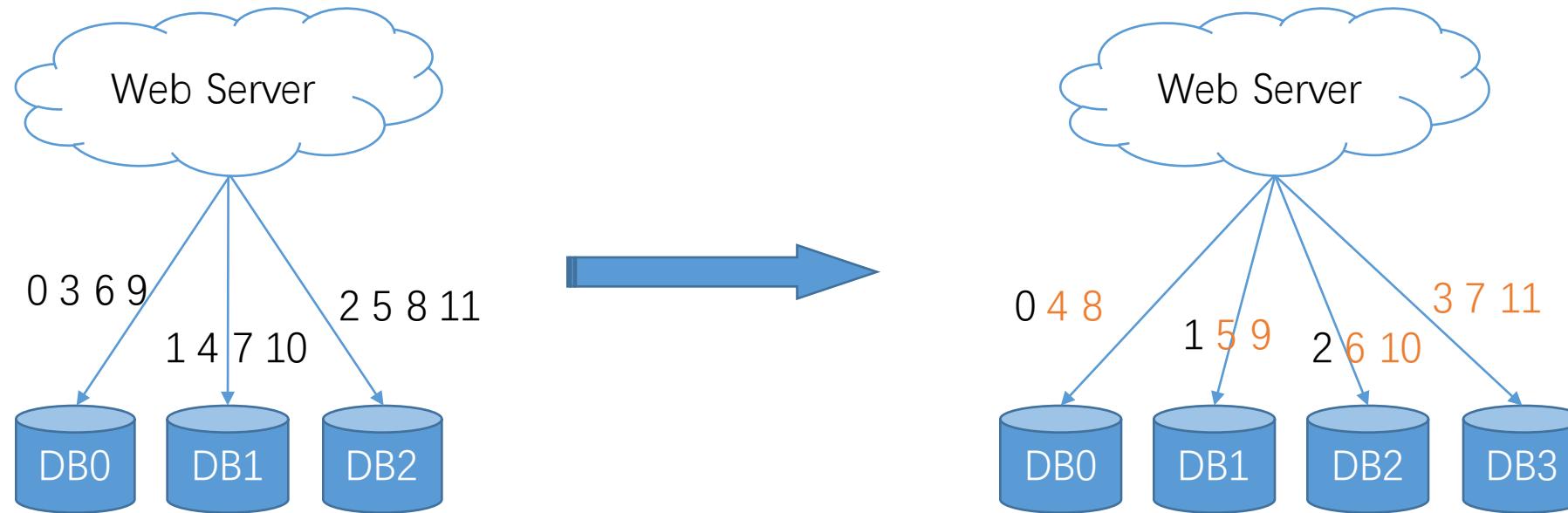
1. 慢，牵一发动全身
2. 迁移期间，服务器压力增大，容易挂
3. 容易造成数据的不一致性

怎么办？

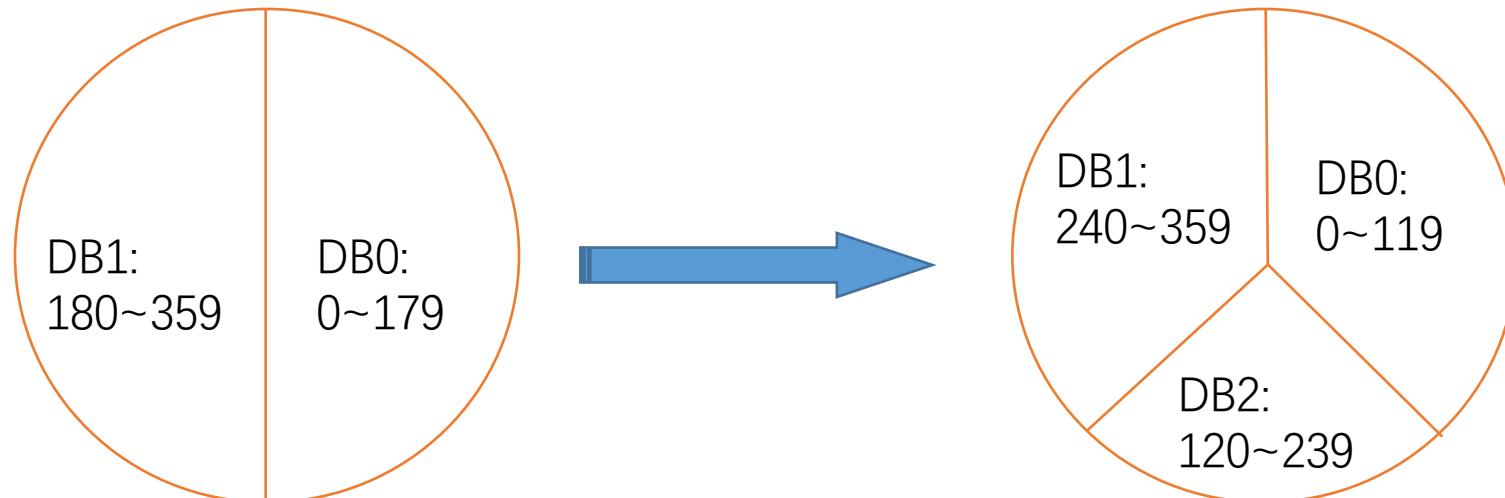
一致性 Hash 算法
Consistent Hashing

Consistent Hashing

- 我们先来说说为什么要做“一致性”Hash
 - $\% n$ 的方法是一种最简单的 Hash 算法
 - 但是这种方法在 n 变成 $n+1$ 的时候，每个 $key \% n$ 和 $\% (n+1)$ 结果基本上都不一样
 - 所以这个 Hash 算法可以称之为：不一致 hash

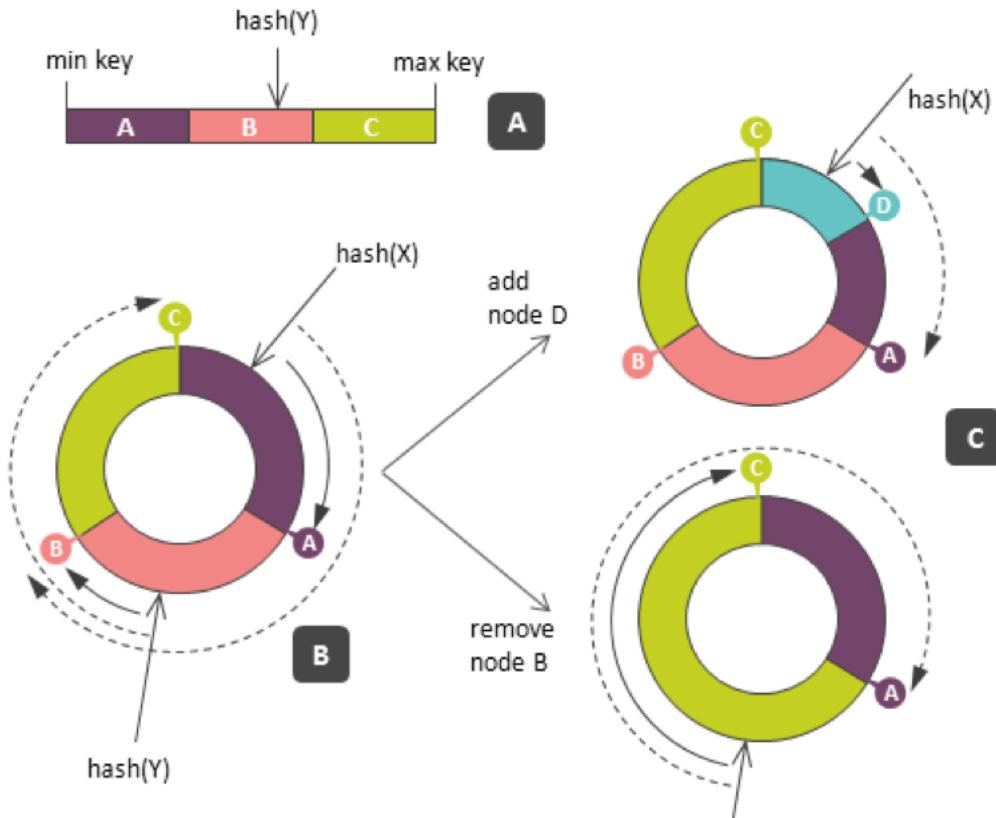


- 一个简单的一致性Hash算法
 - 将 key 模一个很大的数，比如 360
 - 将 360 分配给 n 台机器，每个机器负责一段区间
 - 区间分配信息记录为一张表存在 Web Server 上
 - 新加一台机器的时候，在表中选择一个位置插入，匀走相邻两台机器的一部分数据
- 比如 n 从 2 变化到 3，只有 1/3 的数据移动



Consistent Hashing

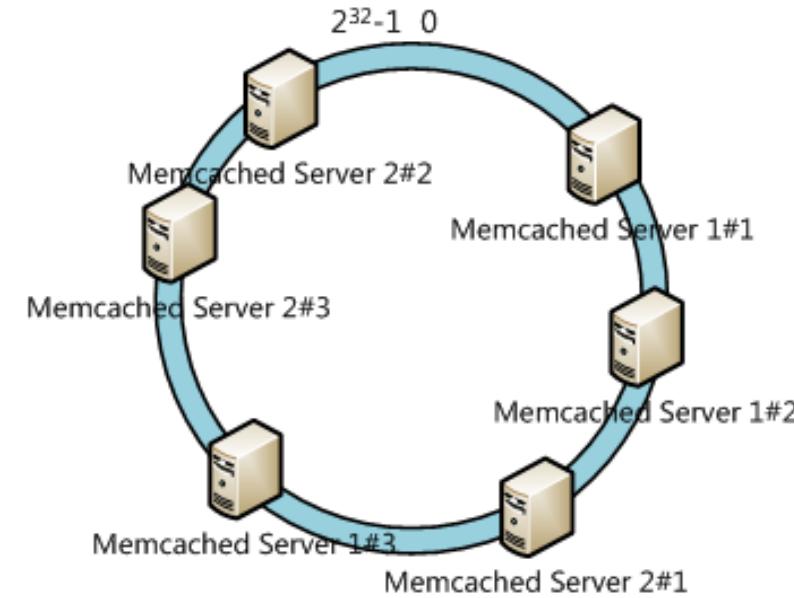
- <http://www.lintcode.com/problem/consistent-hashing/>
- 每一次加入一台新机器
- 只有1台或者2台数据库的数据会被迁移
 - 因为要占据环上的一段区间
- 这是一个比较简单的 Consistent Hashing 的算法
- 提问：这种简单方法中，有什么缺陷？



- 增加一台服务器之后，该新服务器上的数据全从周围的1~2台服务上来
 - 这1~2台服务器的短时间内读压力增大很多，影响到正常服务
- 每次只能从1~2台机器匀数据
 - 数据分配无法做到绝对均匀

Consistent Hashing —— 更实用的方法

- 将整个 Hash 区间看做环
 - 这个环的大小从 0~359 变为 0~ $2^{64}-1$
- 将机器和数据都看做环上的点
- 引入 Micro shards / Virtual nodes 的概念
 - 一台实体机器对应 1000 个 Micro shards / Virtual nodes
- 每个 virtual node 对应 Hash 环上的一个点
- 每新加入一台机器，就在环上随机撒 1000 个点作为 virtual nodes
- 需要计算某个 key 所在服务器时
 - 计算该key的hash值——得到0~ $2^{64}-1$ 的一个数，对应环上一个点
 - 顺时针找到第一个virtual node
 - 该virtual node 所在机器就是该key所在的数据库服务器
- 新加入一台机器做数据迁移时
 - 1000 个 virtual nodes 各自向逆时针的一个 virtual node 要数据



Consistent Hashing

<http://www.lintcode.com/problem/consistent-hashing-ii/>

Replica

MySQL 数据库一般如何做 Replica?

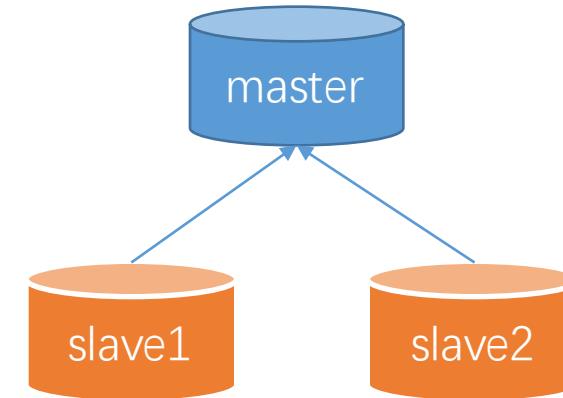
Master - Slave

Master responsible for write / read

Slave responsible for read only

对于大部分系统来说基本够用

- 原理 Write Ahead Log
 - SQL 数据库的任何操作，都会以 Log 的形式做一份记录
 - 比如数据A在B时刻从C改到了D
 - Slave 被激活后，告诉master我在了
 - Master每次有任何操作就通知 slave 来读log
 - 因此Slave上的数据是有“延迟”的
- Master 挂了怎么办？
 - 将一台 Slave 升级 (promote) 为 Master，接受读+写
 - 可能会造成一定程度的数据丢失和不一致



NoSQL 一般如何做 Replica?

重要的数据存三份——顺时针找3台机器

SQL vs NoSQL

SQL需要自己做Sharding, 自己管理Replica

——也就是要写好多代码

而对于NoSQL这一切基本都是自动的

——也就是可以偷懒

- 使用 Cache 优化数据库的读操作
- Session (存在服务器端) / Cookie (存在浏览器端)
- SQL vs NoSQL的一些基本原则
- Friendship 在 Cassandra (NoSQL) 中如何存储
- SQL / NoSQL 数据库如何进行 Sharding
- 一致性 Hash 算法 Consistent Hashing
- SQL 如何进行 Replica – Master-slave
- NoSQL 数据库如何进行 Replica —— 顺时针存三份

Take a break

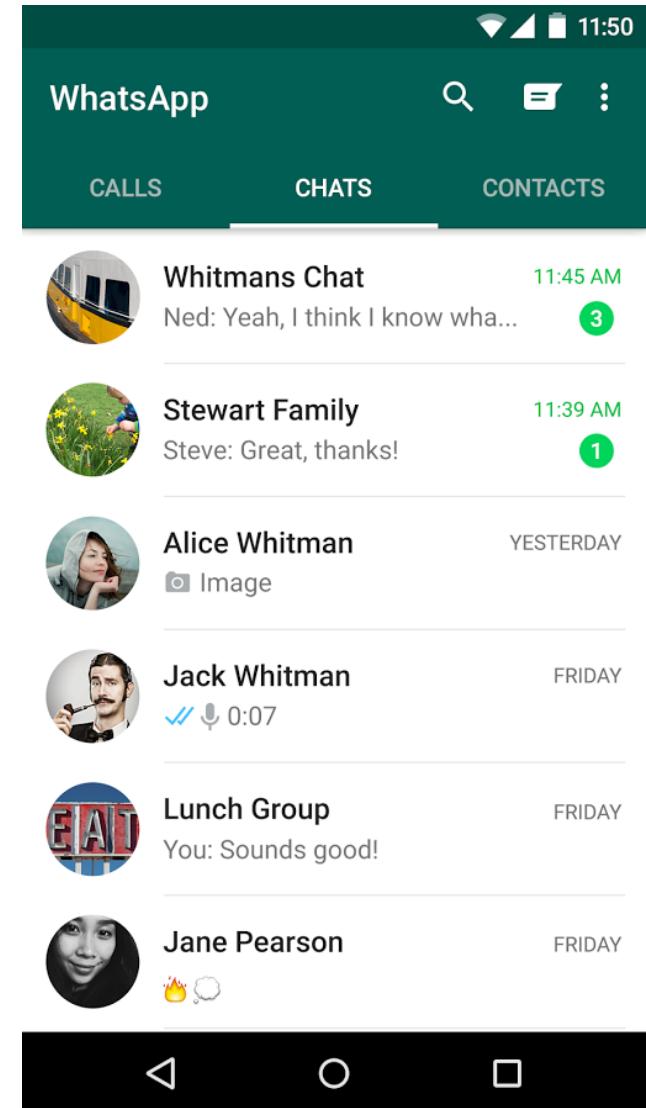
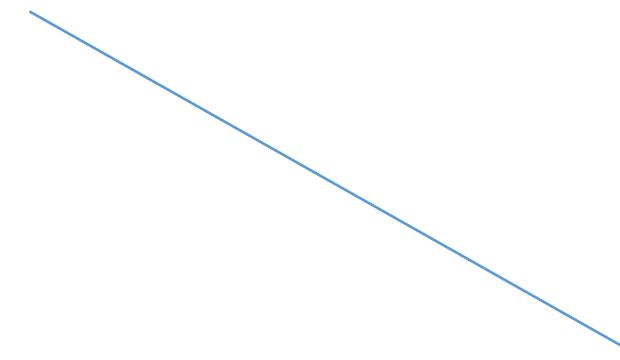
5 minutes

Interviewer: Design Message System

来看看 WhatsApp 最核心的功能怎么设计

- Scenario 场景
 - 发消息
 - 收消息
- Needs 需求
 - 假设平均一个用户每天发20条信息
 - Average QPS = $100M * 20 / 86400 \sim 20k$
 - Peak QPS = $20k * 5 = 100k$
- Application
 - Message Service —— 负责信息管理
 - * Real-time Service —— 负责实时推送信息给接受者

- Message Table (NoSQL)
 - 数据量很大，不需要修改，一条聊天信息就像一条log一样
- Thread Table (SQL) —— 对话表
 - 需要同时 index by
 - Owner User Id
 - Thread Id
 - Participants hash
 - Updated time
 - NoSQL 对multi indexes 的支持并不是很好



Kilobytes 数据 —— Schema

Message Table

message_id	int64	user_id+timestamp
thread_id	int64	
user_id	int64	
content	text	
created_at	timestamp	

Thread Table

user_id	int64	
thread_id	int64	create_user_id+timestamp
participant_ids	text	json
participant_hash	string	avoid duplicate threads
created_at	timestamp	
updated_at	timestamp	index=true

- 用户如何发送消息？
 - Client 把消息和接受者信息发送给 server
 - Server 为每个接受者（包括发送者自己）创建一条 Thread （如果没有的话）
 - 创建一条 message (with thread_id)
- 用户如何接受消息？
 - 可以每隔10秒钟问服务器要一下最新的 inbox
 - 虽然听起来很笨，但是也是我们先得到这样一个可行解再说
 - 如果有新消息就提示用户

- 使用 Cache 优化数据库的读操作
- Session (存在服务器端) / Cookie (存在浏览器端)
- SQL vs NoSQL的一些基本原则
- Friendship 在 Cassandra (NoSQL) 中如何存储
- SQL / NoSQL 数据库如何进行 Sharding
- 一致性 Hash 算法 Consistent Hashing
- SQL 如何进行 Replica – Master-slave
- NoSQL 数据库如何进行 Replica —— 顺时针存三份
- 如何设计 Message System 和构建一个可行的 WhatsApp

Evolve 进化

优化一下刚才的方案

Interviewer: How to Scale?

Message 是 NoSQL, 自带 Scale 属性

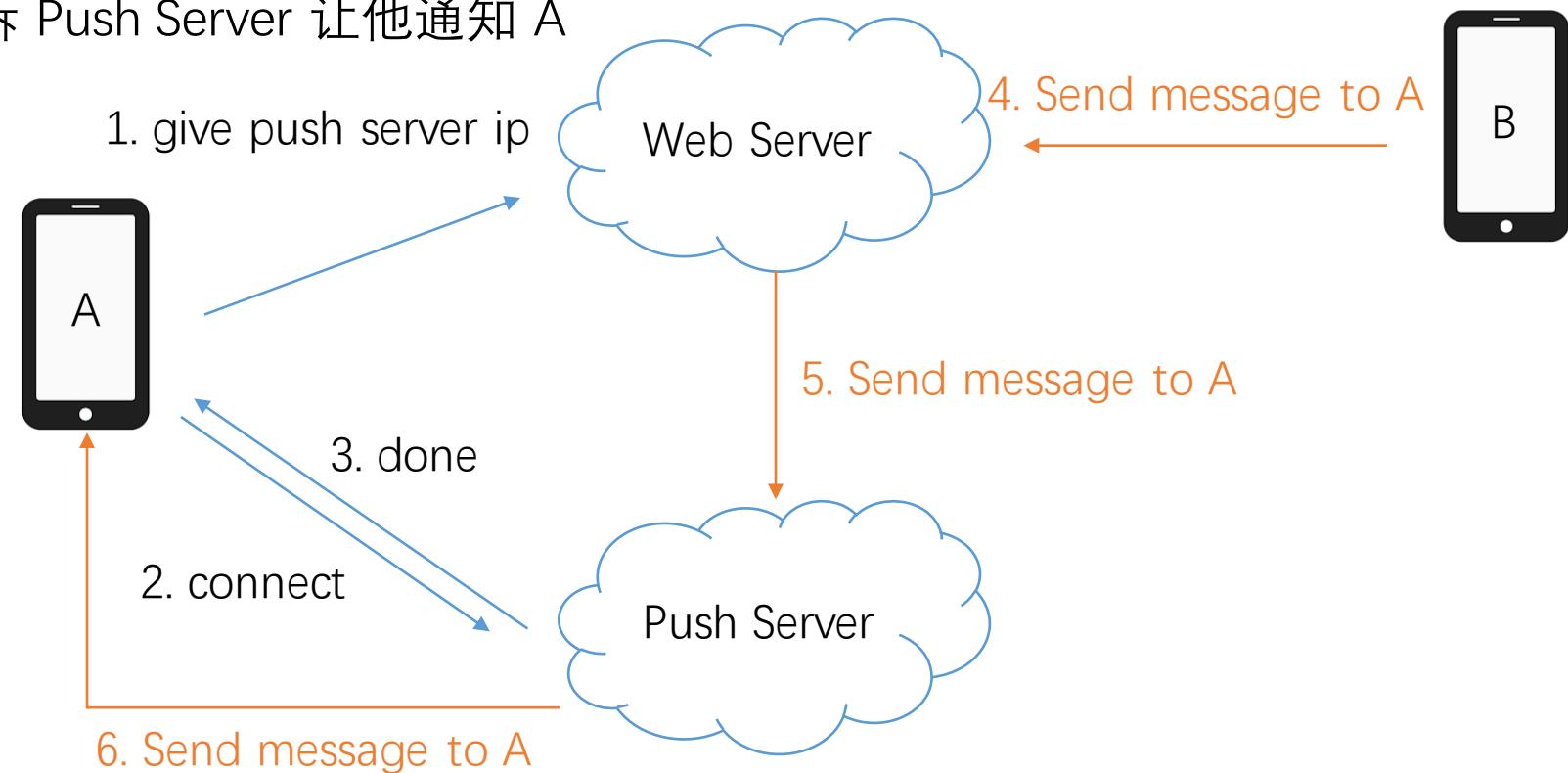
Thread 按照 user_id 进行 sharding

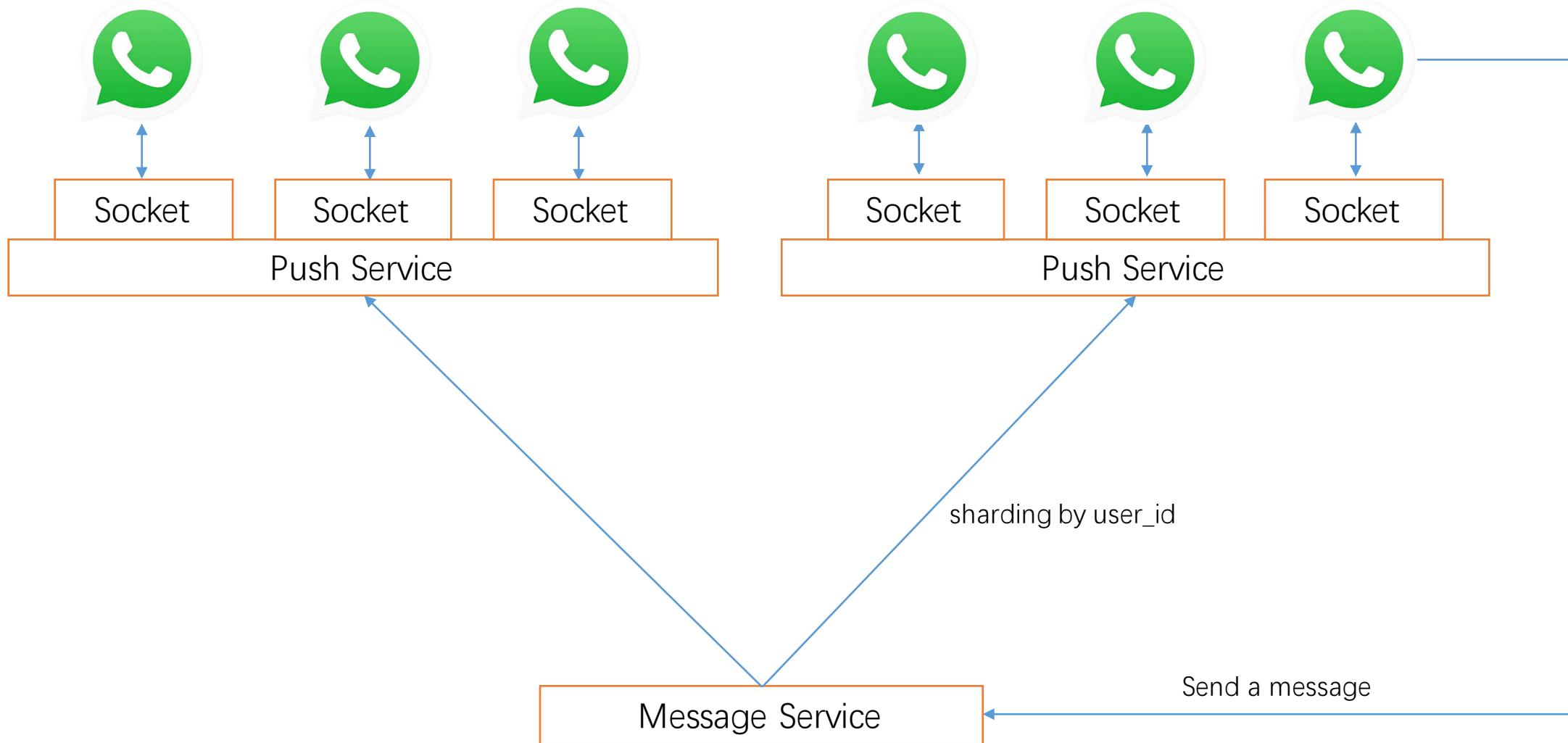
Interviewer: How to speed up?

每隔10秒钟收一次消息太慢了，聊天体验很差，不实时

- 需要引入一个新的概念——Socket
- 再引入一个新的Service —— Push Service
- Push Service 提供 Socket 连接服务，可以与Client保持TCP的长连接
- 当用户打开APP之后，就连接上Push Service 中一个属于自己的socket。
- 有人发消息的时候，Message Service 收到消息，通过Push Service把消息发出去
- 如果一个 用户长期不活跃（比如10分钟），可以断开链接，释放掉网络端口
- 断开连接之后，如何收到新消息？
 - 打开APP时主动Pull + Android GCM / IOS APNS
- Socket 链接 与 HTTP 链接的最主要区别是
 - **HTTP链接下，只能客户端向服务器要数据**
 - **Socket链接下，服务器可以主动推送数据给客户端**

- 用户A打开App后，问 Web Server 要一个 Push Service 的连接地址
- A通过 socket 与push server保持连接
- 用户B发消息给A，消息先被发送到服务器
- 服务器把消息存储之后，告诉 Push Server 让他通知 A
- A 收到及时的消息提醒





Interviewer: How to support large group chat?

支持群聊

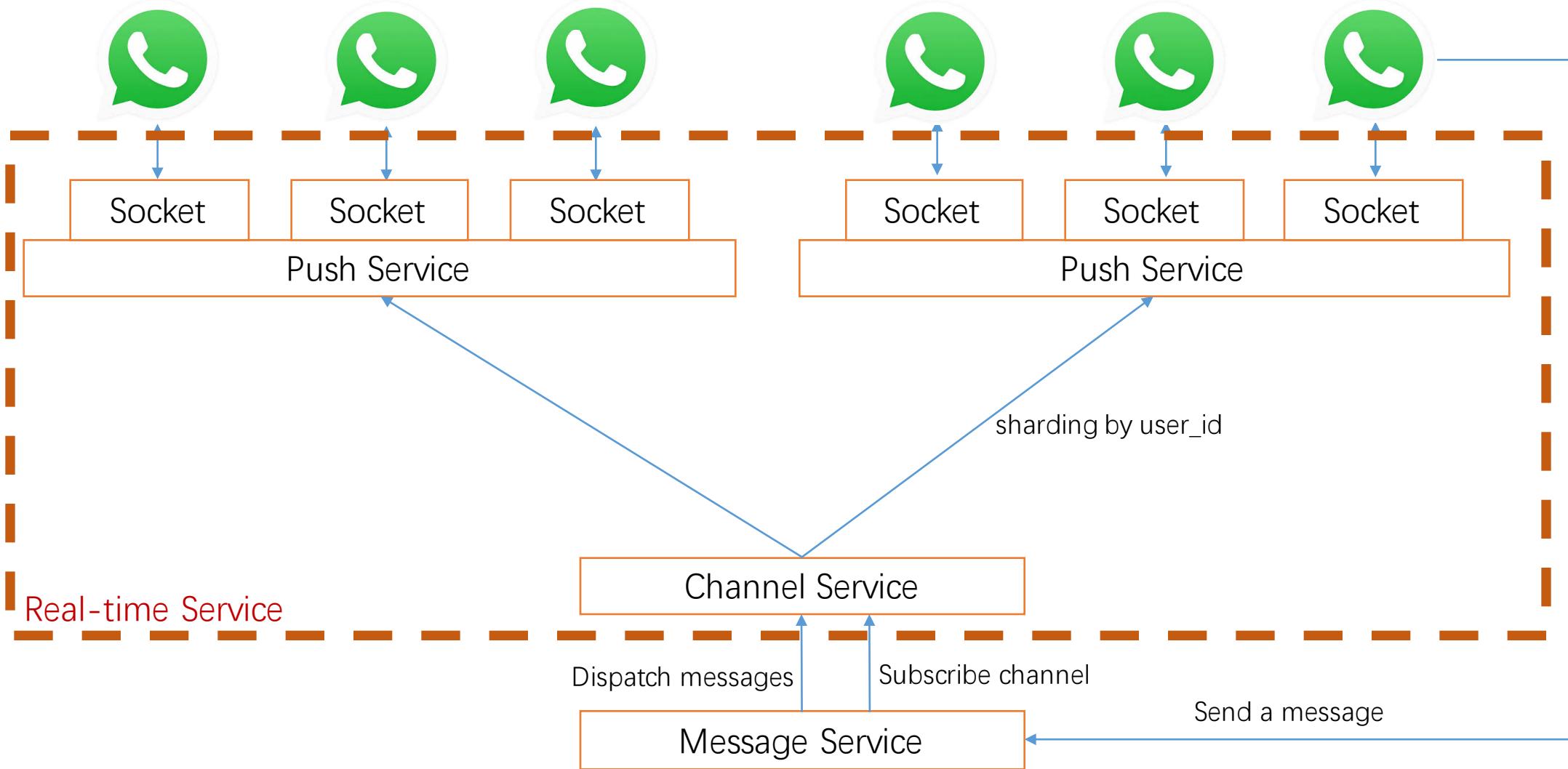


- 问题

- 假如一个群有500人 (1m用户也同样道理)
- 如果不做任何优化，需要给这 500 人一个个发消息
- 但实际上 500 人里只有很少的一些人在线 (比如10人)
- 但Message Service仍然会尝试给他们发消息
 - Message Service (web server) 无法知道用户和Push Server的socket连接是否已经断开
 - 至于 Push Server 自己才知道
- 消息到了Push Server 才发现490个人根本没连上
- Message Service 与 Push Server 之间白浪费490次消息传递

- 解决

- 增加一个Channel Service (频道服务)
- 为每个聊天的Thread增加一个Channel信息
- 对于较大群，在线用户先需要订阅到对应的 Channel 上
 - 用户上线时，Web Server (message service) 找到用户所属的频道（群），并通知 Channel Service 完成订阅
 - Channel就知道哪些频道里有哪些用户还活着
 - 用户如果断线了，Push Service 会知道用户掉线了，通知 Channel Service 从所属的频道里移除
- Message Service 收到用户发的信息之后
 - 找到对应的channel
 - 把发消息的请求发送给 Channel Service
 - 原来发500条消息变成发1条消息
- Channel Service 找到当前在线的用户
 - 然后发给 Push Service 把消息 Push 出去

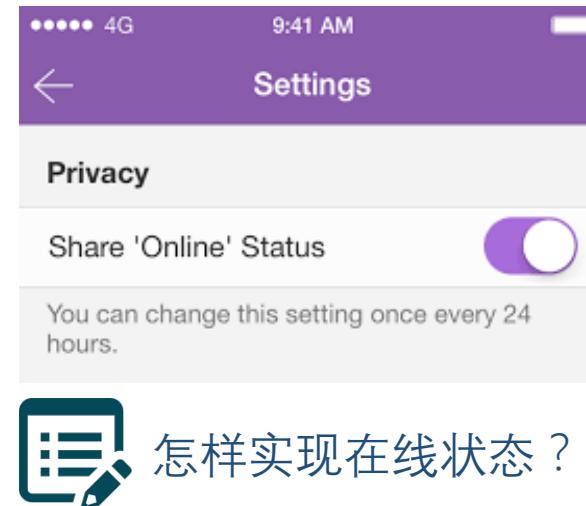


Channel Service用什么存储数据？

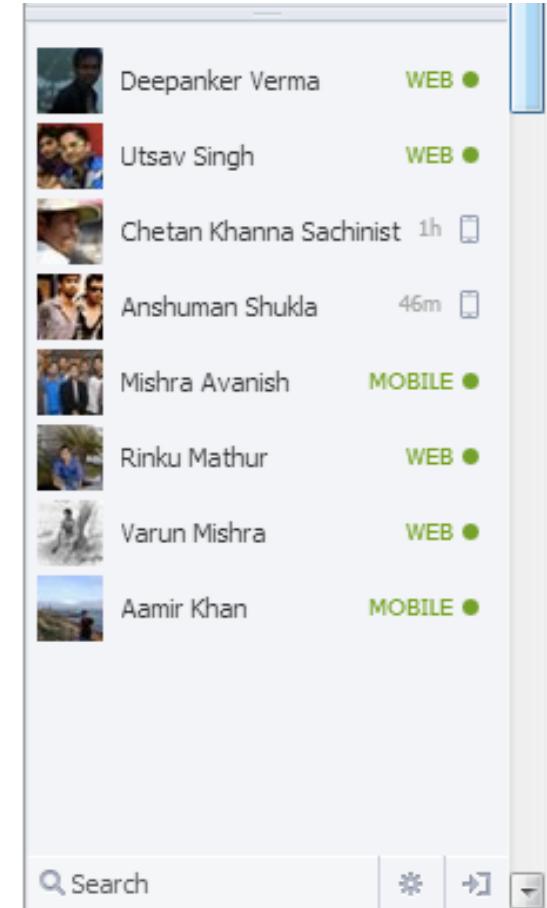
内存就好了，数据重要程度不高，挂了就重启
因为还可以通过 IOS / Android 的 Push Notification 不就

- 使用 Cache 优化数据库的读操作
- Session (存在服务器端) / Cookie (存在浏览器端)
- SQL vs NoSQL的一些基本原则
- Friendship 在 Cassandra (NoSQL) 中如何存储
- SQL / NoSQL 数据库如何进行 Sharding
- 一致性 Hash 算法 Consistent Hashing
- SQL 如何进行 Replica – Master-slave
- NoSQL 数据库如何进行 Replica —— 顺时针存三份
- 如何设计 Message System 和构建一个可行的 WhatsApp
- 引入 Push Service 解决实时性问题
- 引入 Channel service 解决群聊问题

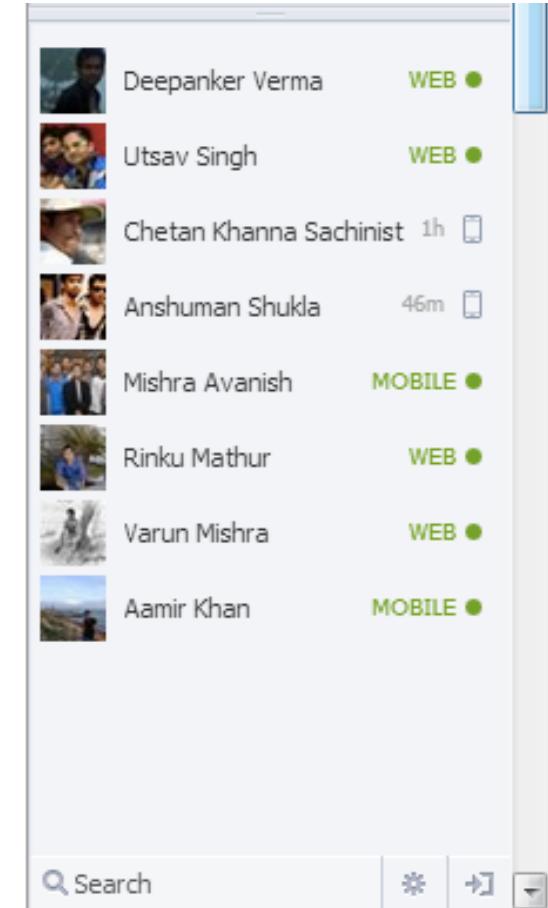
Interviewer: How to check / update online status?



- Update online status 包含两个部分
 - 服务器需要知道谁在线谁不在线 (push or pull ?)
 - 用户需要知道我的哪些好友在线 (push or pull ?)

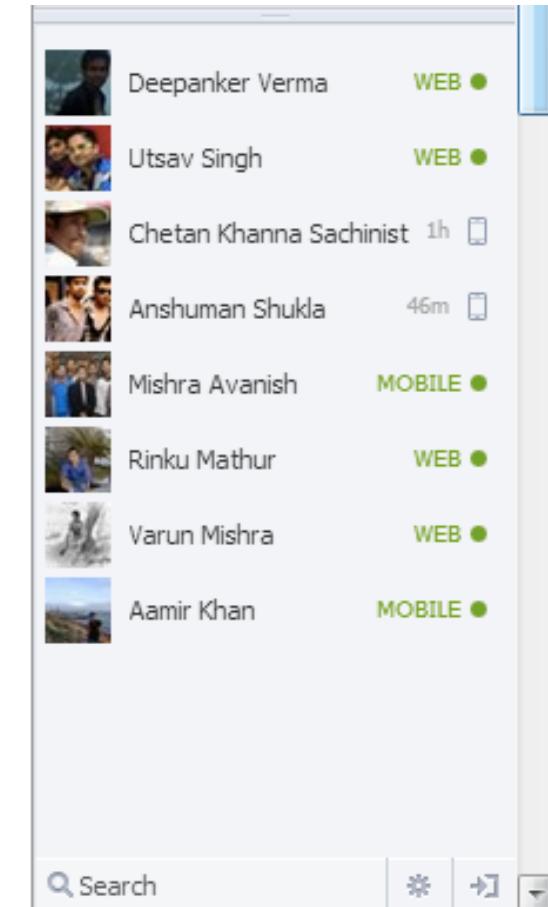


- 告诉服务器我来了 / 我走了
 - 用户上线之后，与 Push Service 保持 socket 连接
 - 用户下线的时候，告诉服务器断开连接
 - 问题：服务器怎么知道你什么时候下线的？万一网络断了呢？
- 服务器告诉好友我来了 / 我走了
 - 用户上线/下线之后，服务器得到通知
 - 服务器找到我的好友，告诉他们我来了 / 我走了
 - 问题1：同上，你怎么知道谁下线了
 - 问题2：一旦某一片区的网络出现具体故障
 - 恢复的时候，一群人集体上线，比如有N个人
 - 那么要通知这N个人的 $N \times 100$ 个好友，造成网络堵塞
 - 问题3：大部分好友不在线



- 告诉服务器我来了 / 我走了
 - 用户上线之后，每隔3-5秒向服务器heart beat一次
- 服务器告诉好友我来了 / 我走了
 - 在线的好友，每隔3-5秒钟问服务器要一次大家的在线状态
- 综合上述
 - 每隔10秒告诉服务器我还在，并要一下自己好友的在线状态
 - 服务器超过1分钟没有收到信息，就认为已经下线
- 可以打开你的 Facebook Messenger 验证一下
 - 打开 console 点击 network
 - 大概每隔3-5秒会pull一次服务器

```
pull?channel=p_1312800249&seq... 200  
3-edge-chat.facebook.com  
  
pull?channel=p_1312800249&seq... 200  
3-edge-chat.facebook.com  
  
pull?channel=p_1312800249&seq... 200  
3-edge-chat.facebook.com
```



- 使用 Cache 优化数据库的读操作
- Session (存在服务器端) / Cookie (存在浏览器端)
- SQL vs NoSQL的一些基本原则
- Friendship 在 Cassandra (NoSQL) 中如何存储
- SQL / NoSQL 数据库如何进行 Sharding
- 一致性 Hash 算法 Consistent Hashing
- SQL 如何进行 Replica – Master-slave
- NoSQL 数据库如何进行 Replica —— 顺时针存三份
- 如何设计 Message System 和构建一个可行的 WhatsApp
- 引入 Push Service 解决实时性问题
- 引入 Channel service 解决群聊问题
- Online Status - Pull vs Push

- 分析出有哪些服务和哪些数据表
- 为每个数据表选择合适的数据存储
- 细化表单结构
- 得到一个 Work Solution
- 知道按照什么 sharding
- 引入 Socket, Push Service, 加速聊天体验
- 引入 Channel Service 解决large group chat问题
- 解决online status update 问题



- Dynamo DB —— 理解分布式数据库(NoSQL)的原理
 - <http://bit.ly/1mDs0Yh> [Hard] [Paper]
- Scaling Memcache at Facebook —— 妈妈再也不担心我的 Memcache
 - <http://bit.ly/1UlpbGE> [Hard] [Paper]
- Consistent Hashing
 - <http://bit.ly/1KhqPEr> [Medium] [Blog]
 - <http://bit.ly/1XU9uZH> [Medium] [Blog]
- 作业：<http://www.lintcode.com/ladder/8/>

- 分布式数据库解决的问题
 - Scalability
- 分布式数据库还没解决很好的问题
 - Query language
 - Secondary index
 - ACID transactions
 - Trust and confidence

	IOPS	Latency	Throughput	Capacity
Memory	(10M)	100ns	10GB/s	100GB
Flash	(100K)	(10us)	1GB/s	1TB
Hard Drive	100	10ms	100MB/s	1TB

	Rack	Datacenter	远距离
P99 Latency	<1ms	1ms	100ms +
Bandwidth	1GB/s	100MB/s	10MB/s -