wDesign User System

4S: Scenario, service, Storage, Scale

Scenario:

* 注册 /登录 /查询/用户信息修改

哪个需求量最大： ！！并不是 登录 （因为很多网站都有保持登陆 这样一个 状态）。查询的话，不仅仅是自己，别人查看发帖子的时候，也会看到你的 avatar，所以query是最大的request

* 支持 100M DAU - Daily Average User
* 注册、登录，信息修改 QPS (Queries per second) 比较小，用 0.1 作为 比例

100M \* 0.1 / 86400 ~ 100 QPS （每秒产生的查询是100个）

* Peak = 100 \* 3 = 300
* 查询的QPS：（一天 查询100次）
  + 100M\* 100 / 86400 ~ 100K ( >> mySQL 可以handle大概 1K Read/Write QPS)
  + 100 = 平均每天每个用户查询信息的操作次数
  + Peak： 300K -- 需要300台mySQL (平均一台 SQL QPS 是 1000 ～ 乐观考虑）

Service: 劝分不劝和，仔细分析

* AuthService: 用来负责登录
* UserService: 用户信息存储与查询
* FriendService： 存储好友相关信息

Storage:

* MySQL/PosgreSQL: 1K QPS 级别 最初级的硬盘型
* MongoDB / Cassandra等硬盘型数据库, NoSQL 数据库的性能 10K QPS （大概10倍mySQL级别）
* Redis （单机）/MemcacheD（分布式） 等内存型NoSQL 数据库性能： 100K - 1Million QPS级别 -- 100K - 1million，几乎是 SQL的query的1000倍。这就是为什么要在 database前面假设 memcached，因为1000次cache hit的时间才等于一次 sql query
* MemcachD 只存在内存，几乎就是内存访问的速度。电后会lost
* Redis 本身是单机 NoSQL，可以用CODIS 豌豆荚的把Redis分布式管理的数据库
* 对于本道问题： 如果只用1台机器，只能用这种级别的数据库

对于每一个系统，判断是读多于写，还是写多于读： UserSystem一定是读非常多，写非常少，读 >> 写 （1000倍）， 需要cache起来 -- 300K 的request，所以用MemCacheD这一类的存储在内存中的

Cache可以理解成Java的HashMap，key value结构

键就是不需要去更慢的介质去读取 -- 比如 Network traffic可能会比file system读取慢 (考虑Browser Cache))

Mem-cacheD: 内存中存储的cache

MemcacheD： 一款负责帮你cache在内存里的软件，像hashmap一样的key value pair。

MemCachD使用方法： cache.set(key, value)

value = cache.get(key)

**cache.set(key, value, ttl = 60)**

value 仅仅存储在内存中60秒

效率高，不保证一定能取到，有可能已经ttl超时delete了

MemcacheD不支持数据持久化

Redis支持数据持久化，但是会慢一点 (TradeOff: 效率高 -- 功能就会少一些)，SQL功能最多，但是也最慢

Cache一定是存在内存中的吗？ 并不是 -- File System也可以做Cache，比如：网络传输就会比File System慢。 **cache的好处是，无需从更慢的 地方去取**。(Cache 的 关MemCacheD 如何优化DB的查询：

1.UserService: 定义两个function

getUser(userID) return obj

if cache 中存在该key，则直接返回 -- 概率 读写概率大概1000 比 1， 99.9%的时间都会进入这个 if statement；只要内存足够大，就可以保证大于 99% 否则， 从DB中读取， 更新cache

setUser(userID, obj) 更新user 信息的时候，必须要先 delete cache，再更新 DB

delete cache （这里不是set cache，或者不是先set database再set cache 为什么？ 为了保证database永远有正确的数据）

db.set(user)（为了保证 数据一致）

考虑失败的场景：

错误写法1：

cache.set(key, user)

// 此时database挂了

database.set(user) -- 失败

此时 cache和database数据不一致， 下次就会从cache中得到了dirty data （因为source of truth应该是database中的data，而不是cache中的数据）

也不能先set database，再delete cache，或先set database，再set cache，总之要保证database是source of truth

所以， 这种写法是错误的

正确写法1：

cache.detele()

// 此时database挂了

database.set()

不会导致 cache和database数据不一致，

关键： 第一步成功， 第二步失败，会不会导致cache和database中数据不一致

Authentication Service: 用户登录

Session table: sessionID, 一个hash值，全局唯一，无规律，有了sessionId就能登录进去，为什么不用user\_id, 因为user\_id 都是很简单的数，如果猜到了user\_id （比如 user\_id = 1 admin）, 就很容易被hack

浏览器 和 server 通信 的一种方式

用户登录后，server 生成一个 session，并返回 session\_key 作为 cookie value 返回给 browser

browser记录cookie，用户每次向server 访问，都会自动带上 该网站的 cookie

Cookie： 令牌，服务器发给Browser作为身份的象征

Server side 在session table生成一个entry，返回给Browser， browser存下狗牌

浏览器插件软件： **Edit This Cookie** --

logout就是删除session key

不同浏览器的session id 是不一样的，这样支持不同浏览器同时登录一个网站

找不着sessionID 就要重新登录（leetcode 就是这么干的）。

* session Table 存在哪里： 如果都存在Cache里，断电了相当于所有用户都logout， 这时所有用户会几乎同时login，login requests 太多了，会把server搞挂。以前的假设就不成立了。所以最好 存 在 数据库里 （最后一次登录在哪里）

User System Summary:

* 写很少，读很多 - （MySQL + MemCacheD）
* 写很少 意味着 一台 MySQL就够了
* 读很多，意味着用 **MemCacheD**进行读操作的优化
* 如果读写操作都很多，可以用Redis这样读写都很快的 支持persistence的cache。

两种cache

-Cache Aside架构: DB和cache不互相沟通，web server决定读/写 DB的数据，还是cache数据

可以认为MemCacheD 是cache aside架构

* Web Server 决定到底是写到 DB 还是 Cache， 。

-Cache Through架构： Web Server只有get.set DB。DB中包含用于persistence的DB和用于做cache (memory) 的部分。 DB来决定是否cache

可以把Redis理解成Cache Through架构

SQL and NoSQL 区别： 一个支持Transaction 一个不支持transaction:

FriendshipService: 单向好友关系: Twitter、instagram，微博

table 中有两个field

* from\_user\_id
* to\_user\_id

双向好友关系： WhatsApp/Facebook,微信

* 可以存储成 A 关注了 B， B 关注了A，两个单向关系
* 也可以存储成 A B 互为关注

好友涉及的操作都很简单: 可以选择NoSQL， 因为Query很简单。

到底选择SQL 还是 NoSQL：

* 大部分情况， SQL和NoSQL都可以。
* 一般来说能用NoSQL就多用 NoSQL，因为性能都很好。
* Transaction： 如果需要支持transaction就不能使用NoSQL，因为NoSQL不支持Transaction。 什么是transaction? 我给你转账了10块钱， 你余额加10，我余额减10，必须同时成功，或同时失败。transaction是用来保证若干条SQL operation能同时成功或同时失败。 SQL原生支持transaction，NoSQL不支持transaction。通过Write Ahead Log来保证Transaction
* 想不想偷懒决定了你想用什么样的SQL：SQL很成熟 （可以存不同数据类型）

NoSQL很多事亲力亲为：

- SQL成熟，有很多数据类型（Integer， timestamp，bool，varchar(20)）

NoSQL需要先Serialization（先把你要存的东西变成字符串）-

- Secondary Index： NoSQL没有实现， select from\_user\_id == ‘A’ and to\_user\_id==’B’,

- 但是NoSQL帮你做了sharding

SQL没有做sharding

database index： 用来查询更快

如果想省服务器，用NoSQL

table中有 from-user-id 和 to-user-ID两个field

e.g. 存双向好友关系 用SQL 有两种方案：

1. 每个entry存双向关系 （smaller user id /bigger user id）: 存储空间小，每次增加或删除只要1次，但是查起来会慢 (Query 语句需要用到SELECT \* from friendship where smaller\_user\_id=’A’ OR ‘bigger\_user\_id’=’A’.。但是这样需要secondary index
2. 每个entry存单向关系，存两次： key = 1, value = 2 && key = 2, value = 1 存储空间更大，每次增加或删除有两次operation，但是读取效率更高 (因为不需要multiple indexes, 读取的时候不需要OR)。方案2的缺点是：
   1. 用了更多硬盘
   2. 增加和删除都是两次操作，当好友关系变化时，写更expensive

对于这道题，因为读比较多，所以方案2比较好

用NoSQL存时，只能用 方案2来存，因为NoSQL不支持 secondary index：拆成两条数据来存： 1关注2， 2关注1

* 第二个方案只需要查一个索引
* 第一个方案要查两个索引 (很多NoSQL不支持secondary index)

SQL vs NoSQL:

* SQL 的 column 数目是有限的
* NoSQL 有 无限多的 column
* SQL 的 column 比如 email，本身没有具体信息
* 但是 NoSQL 的 column key 本身就可以存 a@example.com
* SQL用户信息取出来后，是行式存储，取出来后，一行行就是相关的 data
* NoSQL 是以grid 为单位 -- 某一行/列 的某个格子，

以Cassandra为例剖析典型NoSQL数据结构 (三层结构 row key， column key， value)

* 可以认为： SQL的column是在schema中定好的，有有限个column，**行HANG式存储**
* NoSQL有无限个column，可以随意添加，以某一行某一列决定一个数据

第一层row\_key

* hash\_key, 也就是我们传统中所说的key-value中的key， 一台数据库存不下，某一条数据来了之后 存到 某一个数据库中，最简单 用 row\_key % 3 就能决定这整条数据能存在那一台机器 (sharding --）直接实现了sharding
* 最常用的row\_key: user\_id，因为我们希望每次查询，都尽量在一台机器里返回结果
* row key本身没法进行range query

第二层column\_key （memcached 里面就没有 column key，直接 row key + value)

* 可以进行range query column key 是排好序的:
* 进行排序
* e.g. 查A和B是不是好友，如果只有A作为row key，需要通过A得到A所有的好友list来判断B是否在里面
* 更好的方法是：
  + friend\_id 可以放成column key（比如是 timestamp + frienduserID），比如 row key = A， column Key = B， value doesn’t mattter这个时候可以把value设置为null, 这样就可以通过row key 和column key得到A和B是不是好友
* Query(row\_Key, range\_start, range\_end），Cassandra支持query这个结果 column key的range，但是row key不支持range 查询，因为它是用来指向 哪个 机器的。
* column key 可以是复合值：比如timestamp + user\_id 的组合
* row\_key + column\_key 决定了一个独立的value

第三层value

* 一般来说是string（比如好友关系，比较简单，不需要存啥）
* 如果存很多信息，需要自己serialization

Cassandra支持的操作：

* insert(row\_key, column\_key, value)
* query(rowKey, column\_start, column\_end, )

只包含3个field的数据结构： -- 和mongo db比较。

e.g. 用 casandra 存储 news feed:

row key: owner\_id

column key: <createdAt1, tweet\_id> 用来表示 从新到旧排列

value: tweet\_data

owner\_id -> createdAt+tweetId1

-> createdAt + tweetId2

* 存储newsFeed， row key 是 ownerID， column key是 createdAt and TweetId， value是Feed data

column key是排好序的。

原理： B+ Tree ：多叉树，层树不是特别高

最后一个步骤：

Scale:

* MySQL + MemCacheD 保证了高效读，高效写 （1亿用户 = 1个 MemCaheD + 1 个MySQL）
* 除了QPS还有什么需要考虑（不能有single point of failure）： sharding + replica
  + Sharding数据拆分：将数据拆分成不同部分，保存在不同机器上，这样就算挂也不会导致网站100%不可用。SQL不自带sharding功能，需要码农亲自上手；NoSQL一般都带sharding功能。 怎么拆：发明NoSQL的目的就是 偷懒
    - Vertical Sharding: User Table 放一台数据库，Freidnship Table放一台数据库。 User Table里面有Email Username， Password， avatar。比如头像经常变，但是username不经常变，把它分成两个table。把表格拆成两个 (User Table 和 User Profile Table)，如果profiletable 挂了，不影响User正常登陆。 Vertical Sharding缺点： 当一个table特别大，没法拆。或者
    - Horizontal Sharding: Scale的核心考点！！！！
      * Inconsistent Hashing: 10 台数据库，按照user\_id % 10 进行拆分。这样做的问题： 如果新买了1台机器，原来的%10变成了%11， 几乎所有数据都需要进行大规模迁移，**慢，牵一发而动全身**，迁移过程中，容易造成数据不一致 --- 怎么办: Consistent Hashing
      * 简单的consistent hashing：不要模机器数，模一个固定不变的数，把模出来的数来划分（拿模360为例子，0-179 去 DB0， 180-359去DB1）。这个时候如果加了1个机器，（0-119 DB0， 120-239 DB1， 240-359 DB2），这样的话只有1/3数据被移动。缺点：不均匀。
  + Replica：通常做法是一式三份（重要的事情“”写“”三遍）

两个备份存的很近（美国），第三个存到欧洲。 为什么不三个都很远？通信太慢。 有66%概率恢复很快。