# 1. 通用分析

设计之前需要充分沟通，不需要一开始就照着一个巨牛的方案去设计；

切忌罗列一些关键词；

不要总想着设计一个巨牛的系统，要设计够用的系统（No more no less）；

分析过程比解决方案更重要，通过分析展示知识储备，权衡各种设计方式的利弊；

## 1.1 Scenario 场景

需要涉及哪些功能？性能如何？（Features / QPS / DAU / Interfaces）

## 1.2 Service 服务

将大的服务拆分成小服务（Split / Application / Module ）

## 1.3 Storage存储

数据的存储与访问（Schema / SQL / NoSQL / File System）

## 1.4 Scale扩展

解决缺陷以及可能出现的问题（Sharding / Optimize / Special Case）

# 2. 整体系统设计

## 2.1 Design Twitter

### 2.1.1 Scenario沟通场景

* 需要涉及哪些功能，也可以自己思考
* 需要承受多大的数据访问量
  + - DAU
    - MAU
* 并发用户（Concurrent User）
  + - 日活跃 \* 每个用户平均请求次数 / 一天多少秒 = 150M\*60/86400

~ 100K

* + - 峰值 Peak = Average Concurrent User \* 3 ~300k
    - 快速增长的产品 Fast Growing

Max peak users in 3 months = Peak Users \* 2

* 读频率/写频率

读：300K，写：5K

#### 2.1.1.1 Enumerate 罗列所有可能功能

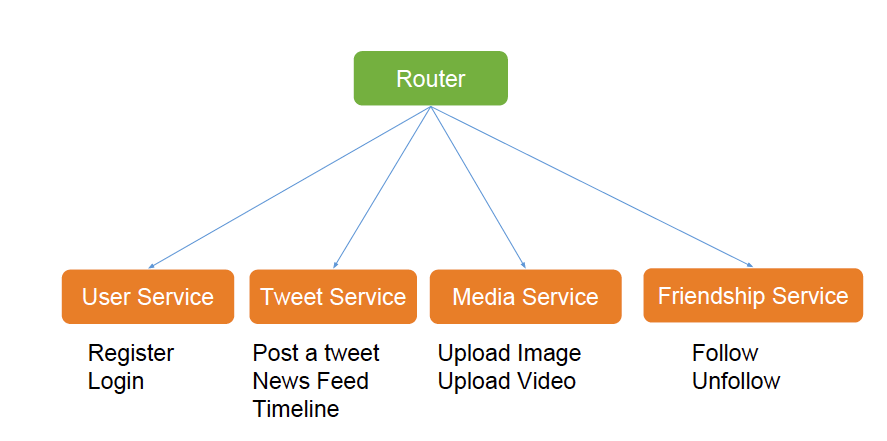
Register/login, Upload Image/video, Post/Share a tweet, Timeline/News Feed

#### 2.1.1.2 Sort 选出核心功能

Post a Tweet, Timeline, News Feed, Follow/Unfollow a user, login

### 2.1.2 Service 设计服务

过一遍所有需求，为每一个需求添加一个服务，并归并相同的服务



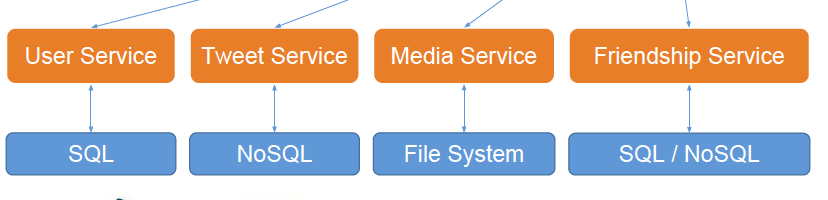
### 2.1.3 Storage存储

设计数据如何存储和访问

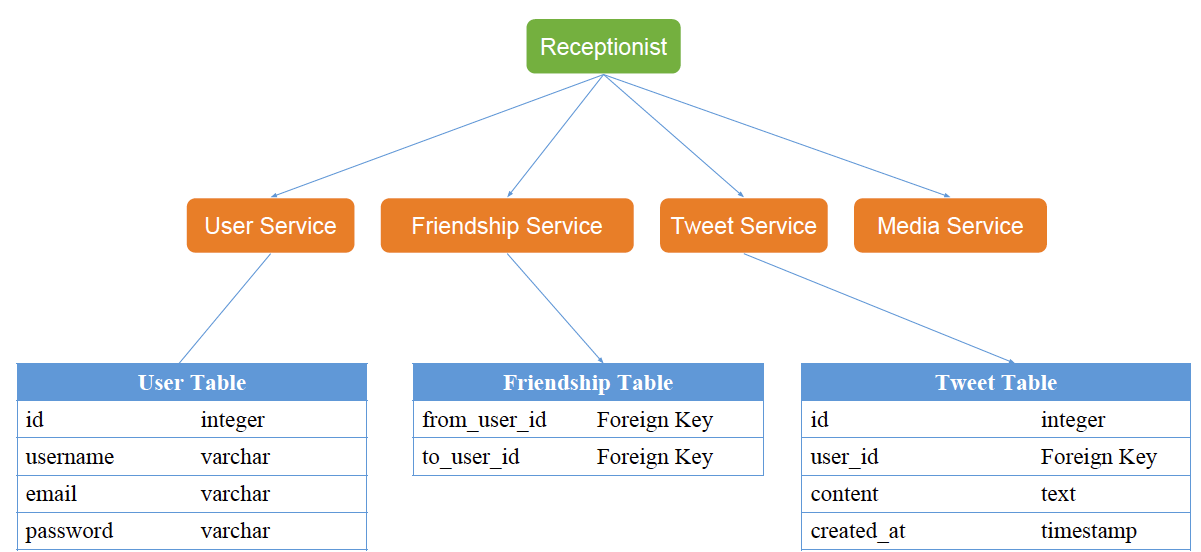
* 关系型数据库（结构化数据）
* 非关系型数据库（结构化数据）
* File System （非结构化数据）

#### 2.1.3.1 实现步骤

步骤1： 为每个Application / Service 选择合适的存储结构



步骤2： 细化数据库表结构



#### 2.1.3.2 News Feed

##### 1. Pull算法

在用户查看News Feed时，获取每个好友的前100条Tweets，合并出前100条News Feed。

复杂度：每个用户有一个Timeline table，用来存储自己的tweet。News Feed => 假如有N个关注对象，则为N次DB Reads的时间 + K路归并时间(可忽略)；Post a tweet => 1次DB Write的时间

**缺点**：用户更新信息的过程中会pending，体验很不好，因为需要同步读取所有关注用户的最新数据；

##### 2. Push 算法

* 为每个用户建一个List存储他的News Feed信息
* 用户发一个Tweet之后，将该推文逐个推送到每个用户的News Feed List中 关键词：Fanout
* 用户需要查看News Feed时，只需要从该News Feed List中读取最新的100条即可

复杂度：需要一个NewsFeed table用来存储所有用户的News Feed，News Feed => 1次DB Read(所有用户的News Feed放在一个表里)；Post a tweet => N个粉丝，需要N次DB Writes（可以异步执行Fanout，无需用户等待）

缺点：Fanout的过程可能会比较慢，尤其是对于有大量follower的情况下，数据更新会比较慢，导致有些用户获得更新会慢，比其他人慢；

##### 3. 相关实现问题

###### 3.1 News Feed Table Sharding

关于 news feed 还是有些不明白的地方. 之前东邪老师说 news feed 是共用同一个 newsfeed table, 然后 user 请求 news feed 时, 再去做 select \* from newsfeed where owner\_id = your\_id limit 100; 之类的.

我的理解是(不确定是否正确), news feed 的上游会有一个 news feed generator, 负责计算一连串的分数计算后, 产生这个 user 可以看到的 news feed, 然后写进 table.

举个例子, 若 user A 和 user B 是朋友, 当 B 发了一篇文, news feed generator 会去决定这篇文章是否该呈现给 A (这边 logic 包括 relevance, blacklist, grouplist 那些), 如果最后算出来决定 A 可以看到, 那 generator 就会把这条 entry (news\_feed\_id, poster\_id, owner\_id, content, timestamp) 写入 newsfeed table

我的疑问是, news feed table 一定会长很快, 所以一定得对他做 sharding. 但想了想, 好像没有比较合适的 sharding 方式.

如果对 news\_feed\_id 做 sharding, 那一个 select request 就要跑所有的机器, 因为每台机器上都有可能有 owner\_id = A 的 entry. 这种方式似乎不 scalable.

如果对 owner\_id 做 sharding, 假设 A 有很多朋友, news feed generator 为 A 写入的 entry 很多, 这样又好像会有机器资料分布不均的问题.

[**Answer: 按照 owner\_id 做sharding, 不会不均匀，因为一个人 “关注” 的对象，不会太多，1000什么的撑死，10万什么的都是spam，会被踢掉的。**]

还是说 news feed 本身不适合用 SQL 解呢? NoSQL 有更好的 approach 吗? 思路有点混乱...

[**Answer: Newfeed 比较适合 NoSQL。当上面你的问题跟sql还是nosql没有太大关系。nosql的情况下，row\_key(hash\_key, partition\_key) = owner\_user\_id, column\_key = post\_id, value = post的基本信息什么的。**]

###### 3.2 Pagination

如果用户看了前100条News Feed里的信息之后还在继续刷新，需要查看第101-200条feed应该如何实现？

问：是不是不管push还是pull模型，如果翻页的话都得pull?

翻页是用户主动操作的过程，所以肯定是由client 发给 server，肯定是一个pull的过程。

问：假设前100条中最早的timestamp是T，就分别请求follow的人在T之前的100条feed，然后再进行合并？

答：对

问：如果恰好有几条feed的timestamp一样该如何处理？

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

通常来说，翻页这个完全可以作为一道单独的系统设计面试题来问你。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，这个时候你再去索引最新的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。

留给你一个思考题：怎么实现往上翻页（刷朋友圈，查看最新的帖子）

###### 3.3 Cache

Twitter 用Pull 模型拿到每个用户的Timelne时，课上老师讲了一种优化是用cache来保存某个用户的最近的100条信息。那么cache之后过了5~10分钟后（或者更长时间），用户再次发送get feed的时候，是不是还是需要访问DB来拿到最近5~10分钟的tweet呢？ **【是的】** 这样的话是不是还是没有解决减少db访问的目的呢？ **【不是】**

答：首先，我们课上说过，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，所以才有此疑问。

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

#### 2.1.3.3 Follow与unfollow

Follow一个用户之后，异步将他的Timeline合并到你的News Feed中；

Unfollow一个用户后，异步将他的Timeline从你的News Feed中删除；

异步的好处在于：用户很快知道Follow / Unfollow是否成功；

异步的坏处在于：操作刚结束的时候，可能在最开始还能看到/看不到他的信息；

#### 2.1.3.4 comment

Tweet table: tweet\_id，user\_id，content, timestamp, like\_nums, comment\_nums, retweet\_nums

Comment table: comment\_id, user\_id, tweet\_id, parent\_id(for nested comment), content, timestamp, like\_nums;

#### 2.1.3.5 Likes

如果不需要显示like/dislike的用户，那么只需要存储一个总数；否则，需要单独设计每个like/dislike记录；NoSQL可以在一个column里存储一个like/dislike list，如果是SQL可以单独一条记录：entity\_id, user\_id, tweet\_id, likes/dislikes, timestamp;

#### 2.1.3.6 好友关系

TBD

### 2.1.4 Scale 扩展

#### 1. Pull的优化

* + DB访问之前加入Cache；
  + Cache每个用户的Timeline；
    - N次DB请求 => N次Cache请求
    - Trade off: Cache最近100条数据？
  + Cache每个用户的News Feed；
    - 优化前：归并N个用户的最近的100挑数据，再取出前100条
    - 优化后：归并 N个用户的在某个时间戳之后的数据；

感想：也可以在用户端显示最新的n条新信息，同时提示用户在上一个timestamp开始到这次timestamp总共有多少个新消息，如果用户想查看所有新消息，额外点击操作获取数据；

#### 2. Push的优化

* 浪费更多存储空间Disk
  + 存储空间便宜
* 不活跃用户 Inactive Users
  + 粉丝排序 Rank followers by weight（for example, last login time）
* 粉丝数目 （Followers）>> 关注数目（Following）Fanout过程可能需要 几个小时！

**面试时的错误回答方案：既然Push不行，就换Pull吧？**

**切忌：想法不坚定，摇摆不定的态度不是一个很好的正确回答方案：**

**正确回答：短期内尝试在现有模型下做最小的改动来进行优化；唱起来看需要评估是否值得换一种更好的模型；**

* 1. 单独开辟一个pool，使用专门的服务器来执行Fanout过程，用于服务这些VIP用户；或者可以在某一个特定高峰期临时增加服务
  2. Push + Pull结合的优化：
     1. 普通用户使用Push；
     2. 对于VIP用户，可以不push数据到Follower的News Feed;
     3. 当Follower需要的时候，来VIP用户的TimeLine Pull数据，合并到News Feed里；

Follow up：Push + Pull结合的方案下如果用户身份转变怎么办？例如掉粉？

* 是否VIP用户和follower的动态变化不绑定，可以定期判定VIP用户身份
* 或者在followers动态变化后，设置缓冲来改变VIP身份

感想：twitter是单向关注关系，所以存在用户有很多followers，如果是类似朋友圈的情况，是一种双向关系，就不会存在某个用户呢存在几十万个好友这样的情况，那么push就很合适；

#### 3. Push vs Pull

|  |  |
| --- | --- |
| **什么时候用Push** | **什么时候用Pull** |
| 资源少 | 资源充足 |
| 实时性要求不高 | 实时性要求高 |
| 用户发帖少 | 用户发帖很多 |
| 双向好友关系（即没有明星问题，例如朋友圈） | 单向好友关系，有明星问题 |

#### 4. 热点事件发生

* 1. **发生**：无数人点赞，评论，转发 ==> 惊群效应（Thundering Herd）
  2. **问题**：同一条数据段时间出现大量的请求；由于是同一条数据，所以什么load balancer，sharding， consistent hashing都不管用；
  3. **解决**：使用Cache，能更好的保护后台的数据库，同时增加数据吞吐效率；
  4. **Follow up 1**： 点赞，转发，评论都会修改这条tweet的基本信息，如何更新？ 即如何更优化的使用Cache？ Cache的使用策略： write through， write back， look aside
  5. **Follow up 2**： Cache失效如何破？因为内存不够或者Cache决策失误，热点信息被提出了Cache，会发生什么？DB会被瞬间收到一大波对该数据的请求，导致数据库挂掉: 解决方案：Facebook Paper：Scaling Memcached in Facebook
  6. **Follow up 3:** 数据库挂了怎么办？用户逐渐增加导致服务器顶不住怎么办？ 请看下回分解；

# 3. 系统功能设计

## 3.1 Rate Limiter

九章网站有文章介绍

# 4. 基本常识

登录时DNS的round robing，返回不同的服务器（Load Balancer）地址；

Browser cache服务器地址，不用每次访问DNS服务器；

访问服务器（Load Balancer），Browser提供cookie，Load Balancer可以根据cookie中的 session id/IP source/account等实现session affinity，定位到唯一的服务器；

如果该唯一的服务器down掉了，user将会被定位到一个新的服务器，session就丢失，一切从头来，如果不想丢失所有以前的状态数据，如何操作？

将session数据存储到一个共享的数据存储（数据库），服务器访问数据存储时，使用memory cache，减少访问数据库次数；

用户被定位到特殊的一台机器上后，需要读取该用户的一些数据，这些数据不是session相关数据，例如用户的信用卡绑定，用户购物车里的保存的物品等，在cookie失效/丢失时这些数据也不能丢失，这些数据都是persist在数据库中的持久化数据，读取这些信息也就需要访问数据库；

Database replication，Load Balancer转发数据到不同的数据库，以此来适应不同的应用环境：read heavy VS write heavy，active passive VS active active

Database partition，主数据库向不同partition的数据库同步数据，不同分区的数据库可以经由不同的Load Balancer来访问，Load Balancer可以通过一些用户信息将请求发送到不同partition的服务器上；

Single Point Failure，任何可能出现的单独一个应用服务器，数据库服务器，Load Balancer都是潜在的单点失败；

# 5. 分布式系统

ACID CAP 幂等性(Idempotency)

一个分布式系统里面，节点组成的网络本来应该是连通的。然而可能因为一些故障，使得有些节点之间不连通了，整个网络就分成了几块区域。数据就散布在了这些不连通的区域中。这就叫分区。

当你一个数据项只在一个节点中保存，那么分区出现后，和这个节点不连通的部分就访问不到这个数据了。这时分区就是无法容忍的。提高分区容忍性的办法就是一个数据项复制到多个节点上，那么出现分区之后，这一数据项就可能分布到各个区里。容忍性就提高了。

然而，要把数据复制到多个节点，就会带来一致性的问题，就是多个节点上面的数据可能是不一致的。要保证一致，每次写操作就都要等待全部节点写成功，而这等待又会带来可用性的问题。

总的来说就是，数据存在的节点越多，分区容忍性越高，但要复制更新的数据就越多，一致性就越难保证。为了保证一致性，更新所有节点数据所需要的时间就越长，将数据复制到所有节点，导致可用性出现问题。

例如有五个节点：n1~n5 ，出现网络分区被分成两组：[n1~n2]和[n3~n5]，那么当n1出来客户端请求时（为了处理这种情况，也就是说"容忍网络分区"，即支持 P）：

1. 如果要保证C(一致性)，那么它需要把消息复制到所有节点，但是网络分区导致无法成功复制到n3~n5，所以它只能返回"处理失败"的结果给客户端。（这时系统就处于不可用状态，**即丧失了A**）
2. 如果要保证可用性A，那么n1就只能把消息复制到n2，而不用复制到n3~n5(或者无视复制失败/超时)，但n3同时也可能在处理其他客户端的请求(譬如对同一个值进行修改)，n3也为了保证A而做了同样的处理。 那么 [n1~n2]和[n3~n5]的状态就不一致了，**于是就丧失了 C**

# 6. 缓存相关

缓存穿透，即大量不命中情况，如何保护缓存：见后续；

缓存踩踏，即缓存失效/过期情况下，如何保护存储：见后续；

热点事件下，短时间内大量请求访问统一数据：基于缓存策略来解决（write-through、write-back、look-aside）；

缓存系统crash：保证缓存层的高可用性，一致性哈希下多实例使用；

缓存扩容，惊群效应（Thundering Herd）：一致性哈希解决；

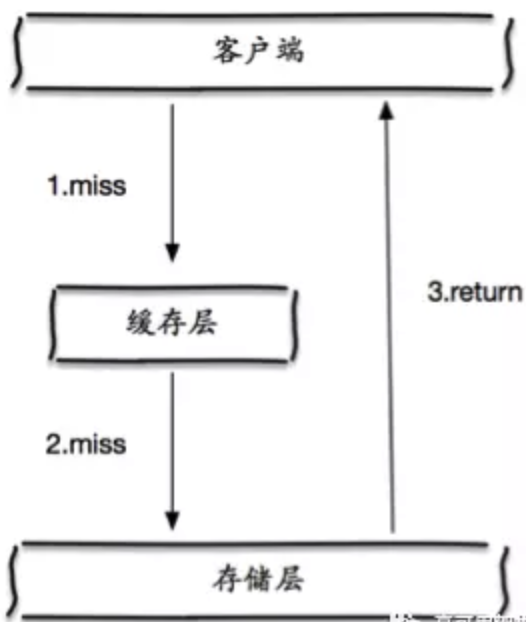
缓存的存在就是为了提高用户的访问速度，同时也是保护后端的存储服务器；

## 6.1. 缓存穿透预防及优化

缓存穿透是指查询一个根本不存在的数据，缓存层和存储层都不会命中，但是出于容错的考虑，如果从存储层查不到数据则不写入缓存层，如图 11-3 所示整个过程分为如下 3 步：

* 缓存层不命中
* 存储层不命中，所以不将空结果写回缓存
* 返回空结果

缓存穿透将导致不存在的数据每次请求都要到存储层去查询，失去了缓存保护后端存储的意义。

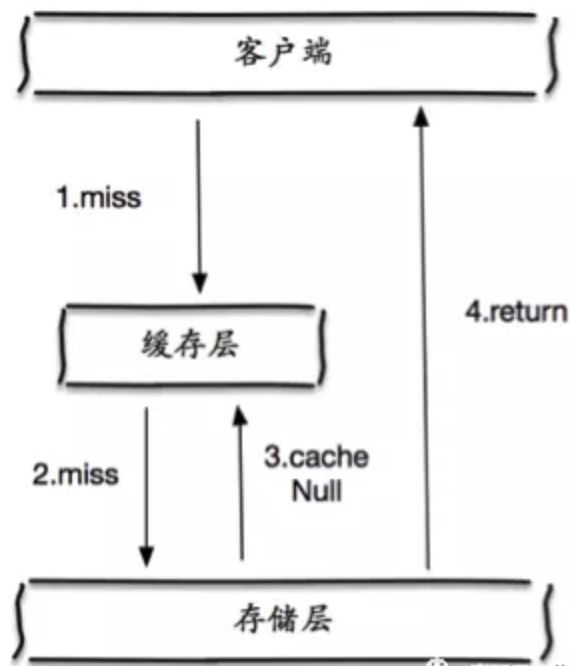


缓存穿透问题可能会使后端存储负载加大，由于很多后端存储不具备高并发性，甚至可能造成后端存储宕掉。通常可以在程序中分别统计总调用数、缓存层命中数、存储层命中数，如果发现大量存储层空命中，可能就是出现了缓存穿透问题。

造成缓存穿透的基本有两个。1.业务自身代码或者数据出现问题; 2.一些恶意攻击、爬虫等造成大量空命中，下面我们来看一下如何解决缓存穿透问题。

#### 1.1 缓存空对象

如下图所示，当第 2 步存储层不命中后，仍然将空对象保留到缓存层中，之后再访问这个数据将会从缓存中获取，保护了后端数据源。



缓存空对象会有两个问题：

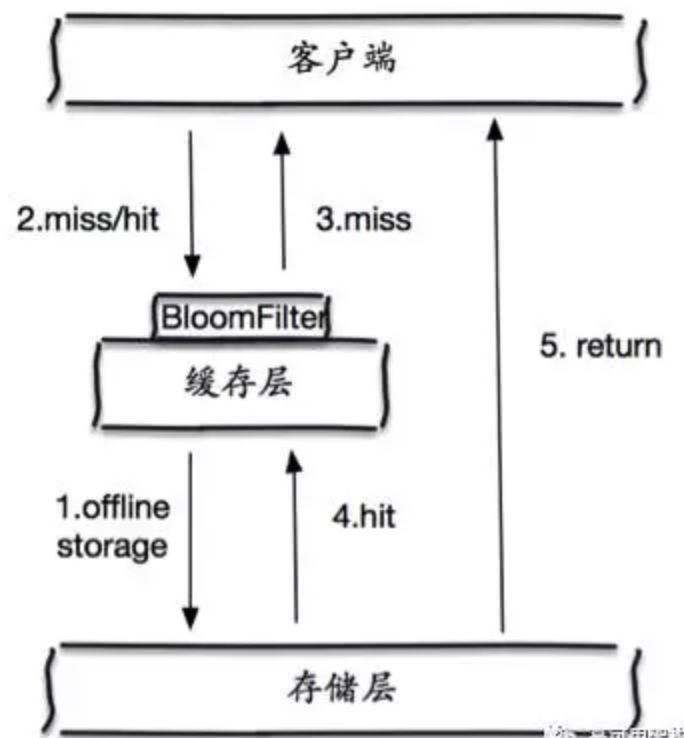
第一，空值做了缓存，意味着缓存层中存了更多的键，需要更多的内存空间 ( 如果是攻击，问题更严重 )，比较有效的方法是针对这类数据设置一个较短的过期时间，让其自动剔除。

第二，缓存层和存储层的数据会有一段时间窗口的不一致，可能会对业务有一定影响。例如过期时间设置为 5 分钟，如果此时存储层添加了这个数据，那此段时间就会出现缓存层和存储层数据的不一致，此时可以利用消息系统或者其他方式清除掉缓存层中的空对象。

|  |
| --- |
| String get(String key) {  String cacheValue = cache.get(key);  If(cacheValue == null) {  String storageValue = storage.get(key);  cache.set(key, storageValue);  if(storageValue == null) {  cache.setExpire(key, 60\*5);  }  return storageValue;  } else {  return cacheValue;  }  } |

#### 1.2 布隆过滤器拦截

如下图所示，在访问缓存层和存储层之前，将存在的 key 用布隆过滤器提前保存起来，做第一层拦截。例如： 一个个性化推荐系统有 4 亿个用户 ID，每个小时算法工程师会根据每个用户之前历史行为做出来的个性化放到存储层中，但是最新的用户由于没有历史行为，就会发生缓存穿透的行为，为此可以将所有有个性化推荐数据的用户做成布隆过滤器。如果布隆过滤器认为该用户 ID 不存在，那么就不会访问存储层，在一定程度保护了存储层。



这种方法适用于数据命中不高，数据相对固定实时性低（通常是数据集较大）的应用场景，代码维护较为复杂，但是缓存空间占用少。

#### 1.3 两种方案对比

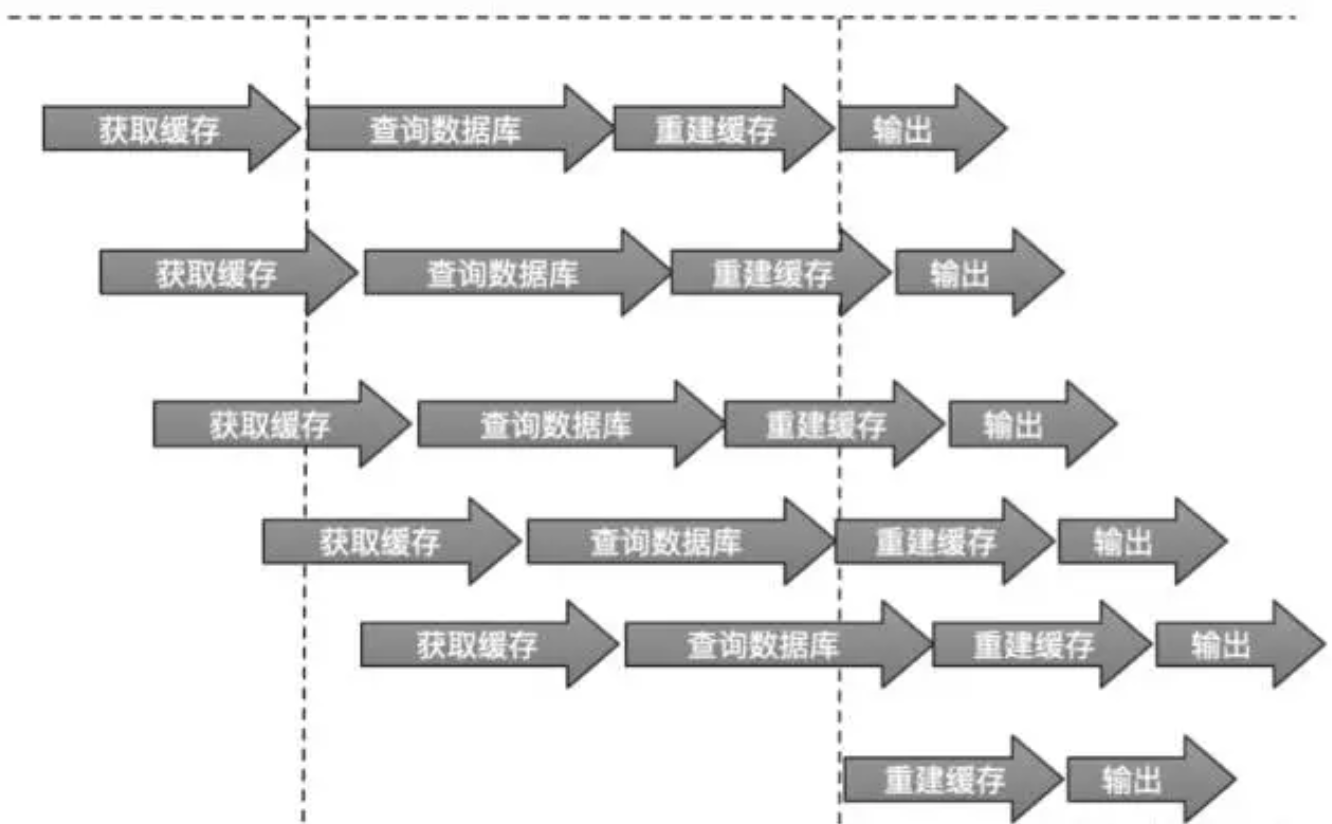
|  |  |  |
| --- | --- | --- |
| **解决方案** | **使用场景** | **维护成本** |
| 缓存空对象 | 1. 数据命中不高  2. 数据频繁变化实时性高 | 代码维护简单  需要过多缓存 |
| 布隆过滤器 | 1. 数据命中不高  2. 数据相对固定实时性低 | 代码维护比较复杂  缓存空间使用较少 |

## 6.2. 缓存热点 key重建优化

开发人员使用缓存 + 过期时间的策略既可以加速数据读写，又保证数据的定期更新，这种模式基本能够满足绝大部分需求。但是有两个问题如果同时出现，可能就会对应用造成致命的危害：

* 当前 key 是一个热点 key( 例如一个热门的娱乐新闻），并发量非常大。
* 重建缓存不能在短时间完成，可能是一个复杂计算，例如复杂的 SQL、多次 IO、多个依赖等。

在缓存失效的瞬间，有大量线程来重建缓存 ( 如下图)，造成后端负载加大，甚至可能会让应用崩溃，Cache stampede情况。

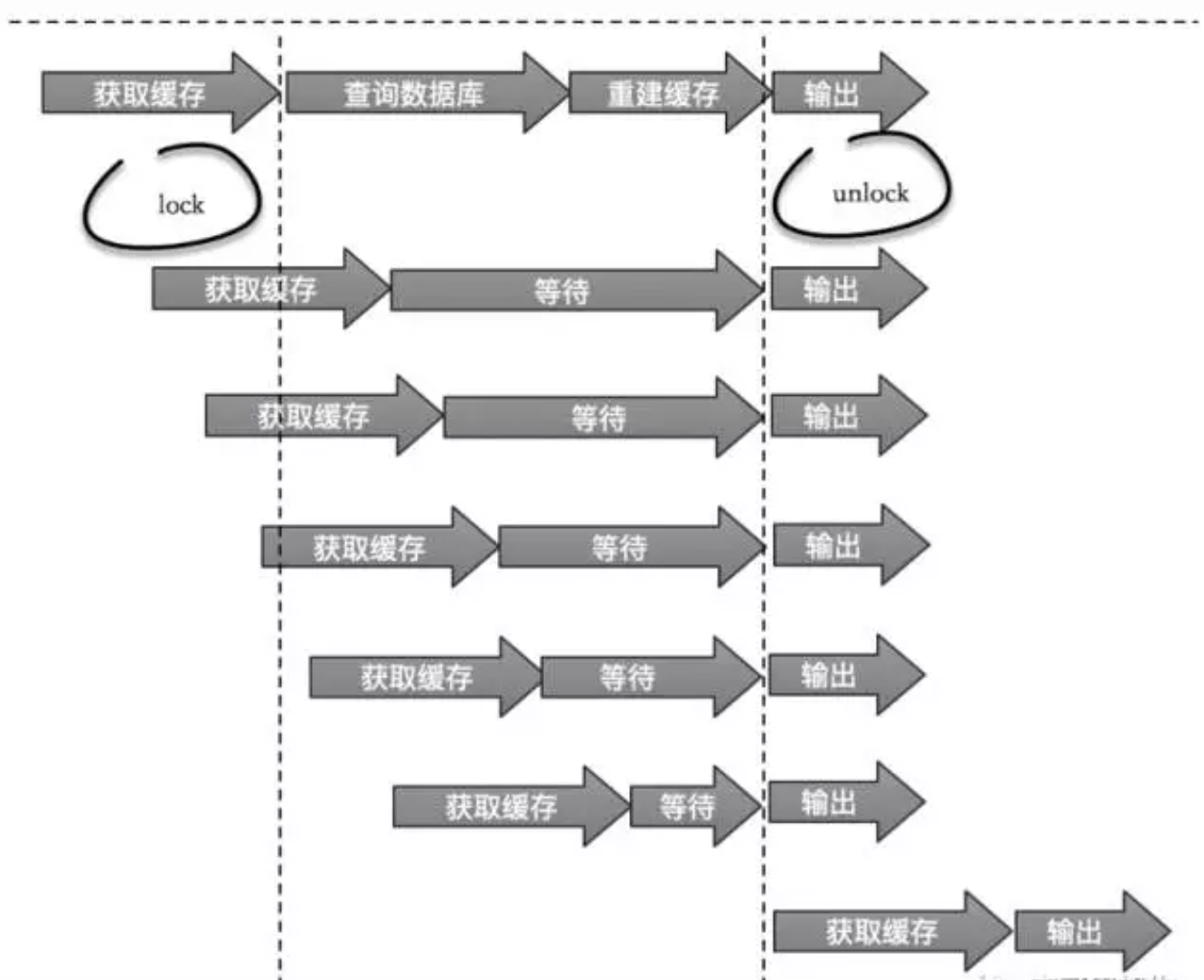


要解决这个问题也不是很复杂，但是不能为了解决这个问题给系统带来更多的麻烦，所以需要制定如下目标：

* + 减少重建缓存的次数
  + 数据尽可能一致
  + 较少的潜在危险

#### 2.1 互斥锁（mutex key）

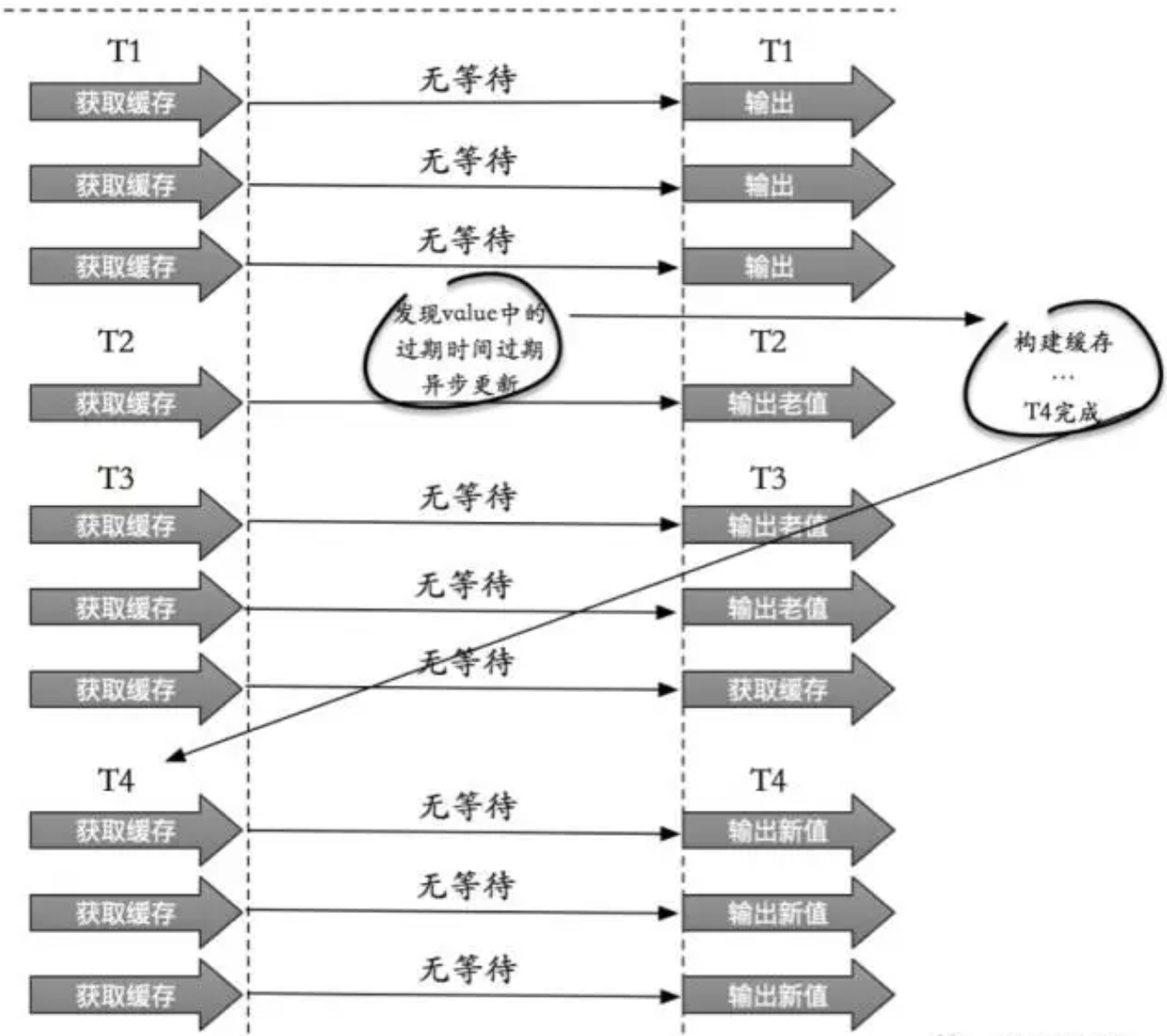
此方法只允许一个线程重建缓存，其他线程等待重建缓存的线程执行完，重新从缓存获取数据即可，整个过程如图



#### 2.2 永不过期

“永远不过期”包含两层意思：

* 从缓存层面来看，确实没有设置过期时间，所以不会出现热点 key 过期后产生的问题，也就是“物理”不过期。
* 从功能层面来看，为每个 value 设置一个逻辑过期时间，当发现超过逻辑过期时间后，会使用单独的线程去构建缓存。



从实战看，此方法有效杜绝了热点 key 产生的问题，但唯一不足的就是重构缓存期间，会出现数据不一致的情况，这取决于应用方是否容忍这种不一致。

#### 2.3 方案对比

作为一个并发量较大的应用，在使用缓存时有三个目标：

1. 加快用户访问速度，提高用户体验。
2. 降低后端负载，减少潜在的风险，保证系统平稳。
3. 保证数据“尽可能”及时更新。

|  |  |  |
| --- | --- | --- |
| **解决方案** | **优点** | **缺点** |
| 简单分布式锁 | 简单，保证一致性 | 1. 代码复杂度大 2. 存在死锁风险 3. 存在阻塞风险 |
| 永不过期 | 基本热点key问题 | 1. 不保证一致性 2. 逻辑过期时间增加代码维护成本和内存成本 |

## 6.3. 缓存更新策略

#### 3.1 Write-through

Data is written to the cache and the backing store location at the same time.

#### 3.2 Write-around

Data is written only to the backing store without writing to the cache.

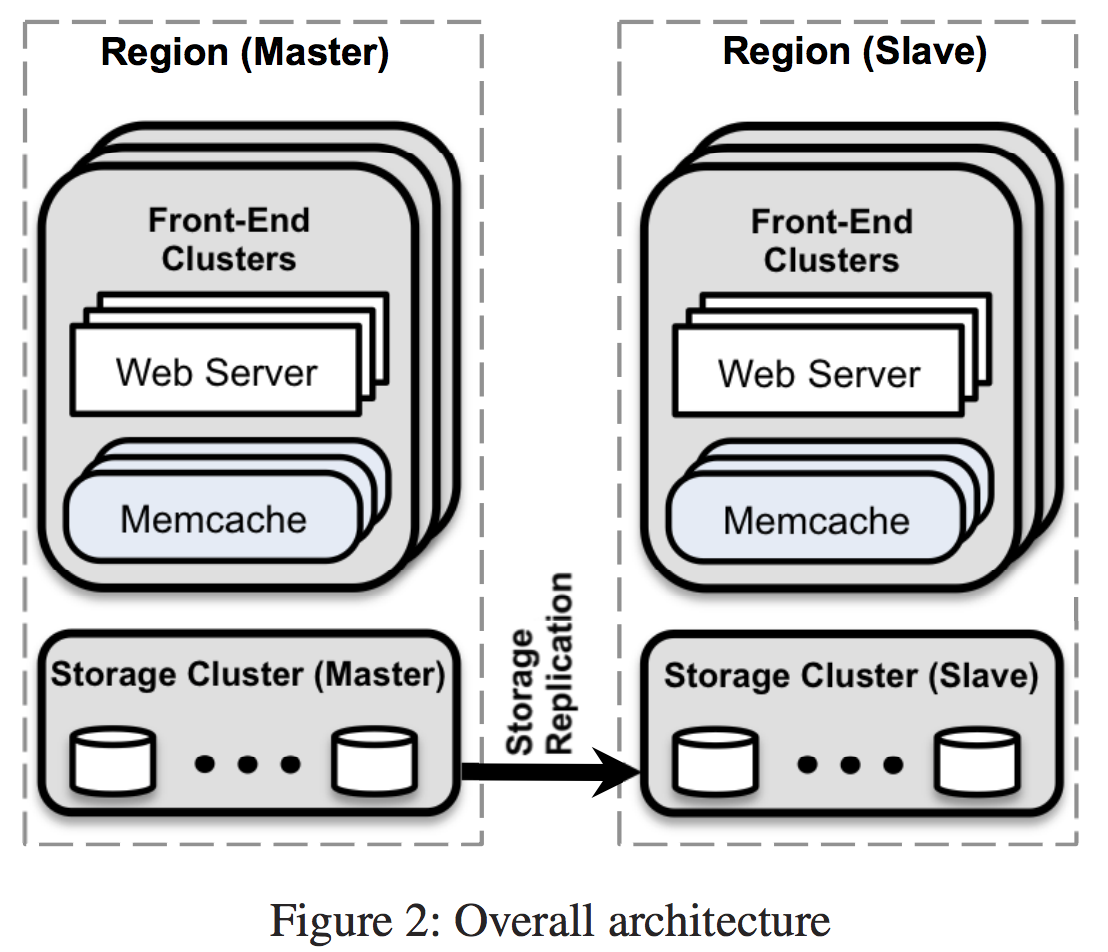
#### 3.3 Write-back

Data is written only to the cache without writing to the backing store.

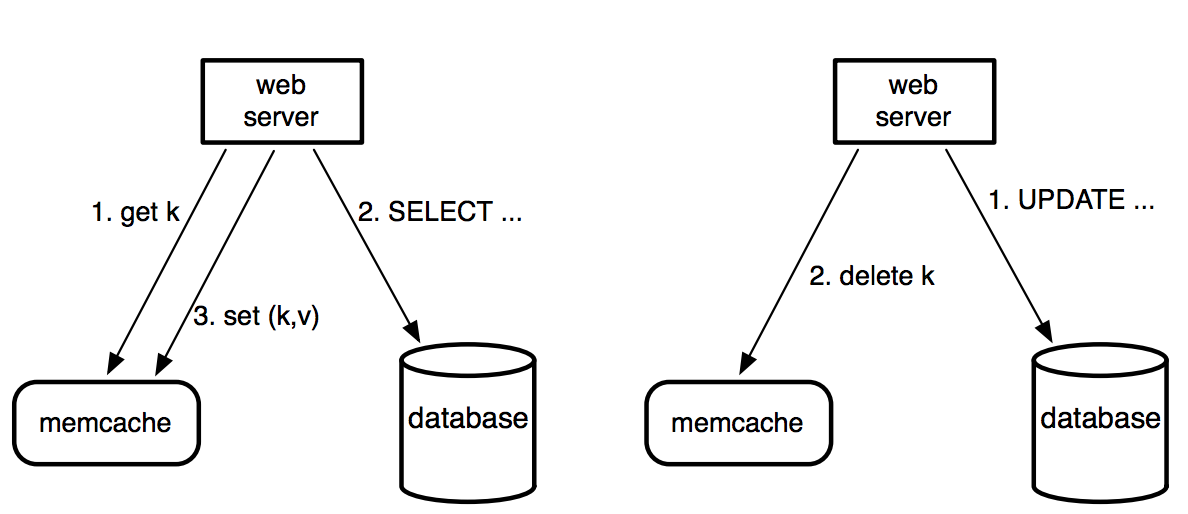
#### 3.4 方案比较

|  |  |  |  |
| --- | --- | --- | --- |
| 解决方案 | 优点 | 缺点 | 适用场景 |
| Write-through | Ensure fast retrieval while making sure the data is in the baking store. | Write data will experience latency. | It’s good for applications that write and then re-read data frequently. |
| Write-around | Not flood the cache with data that may not be subsequently re-read. | Reading recently written data will result in a miss. | It’s good for applications that will not re-read the recently written data. This will result in lower write latency but higher read latency. |
| Write-back | Low latency and high throughput for write-intensive applications. | There is data availability risk before the data is persisted to the backing store. | It’s the best performer for both read and write. |

## 6.4. Scalling memcache at Facebook



Look-aside/Write-through



### 1. In Custer

#### 1.1 Reduce Latency to Cache Layer

* Parallel requests and batching
* Communication between client and Server: UDP vs TCP

Get is using UDP and is provided as a local library. Set is using TCP and is provided as a standalone process (daemon).

* Congestion control in case of in-cast congestion

#### 1.2 Reduce Load to Persistence Layer

* Lease mechanism: stale data and herding issue
* Use different cache pool based on different data access pattern

#### 1.3 Error handling

* User “Gutter” cache pool to send request when client receives no response.

### 2. In Region

Why split web servers and cache server into different front clusters? Simply scale web servers and cache servers will incurs the following problems:

* + - * The total connections between web servers and cache servers is N^2.
      * Incase congestion
      * The cache server which hold the hotkey may be overloaded.

#### 2.1 Regional Invalidation

Storage layer is responsible for invalidating cached data. As an optimization, a web server that modifies data also send invalidations to its own cluster.

All the invalidation requests sent from Web server or storage layer will be sent to the Memcached client.

#### 2.2 Regional Pool

* Use different pool for the data with low access rate
* “Gutter” pool for regional pool

#### 2.3 Code Cluster Warm

The cold cluster which has empty cache will fetch data from a warm cluster instead of storage layer.

Care must be taken to avoid inconsistencies due to race condition:

* + - 1. Client A in cold cluster delete a data;
      2. Client B in cold cluster fetch data from a warm cluster and insert the data in the code cluster;
      3. All the other clusters receive the invalidation requests. The data is inconsistent in current cold cluster.

Solution: By default, all deletes to the cold cluster are issued with a two second hold-off. When client B fetch the data and insert it into the cold cluster, the add will fail.

### 3. Across Region : Consistency

#### 3.1 From Master Region

* Master Region A + Cluster A:

Web server updates data in database and notifies daemon to delete the key in cache.

* Master Region A + Cluster B/C/D:

Persistence layer notify other clusters’ daemons to delete the key in cache.

* Slave Region B/C/D:

Persistence layer/MySQL copy data from master to slave. The salve region’s persistence layer will notify all the clusters’ daemons to delete the key in cache.

#### 3.2 From Slave Region

Slave Region B + Cluster B: (Update Key: k)

1. Create a flag fk for key k and put fk and k into regional pool;
2. Send “fk + k + SQL” to master region’s persistence layer;
3. Delete data in cash in cluster B;

For the subsequent requests for k, a data miss occurs. Then check if “fk” exists in regional pool. The request will be redirected to a master or local region depend on the presence of “fk”. (Trade the latency for a decreased probability of reading stale data)

Master Region A + All clusters in master region:

1. Execute the SQL and update data;
2. Notify clusters’ daemon to delete the data in cache;
3. Copy data to all slave regions;

Other Slave Regions: (Update Key: k)

1. Persistence layer notify all clusters’ daemon to delete data;

Slave Region B:

1. Delete “fk + k” from regional pool;
2. Persistence layer notify all clusters’ daemon to delete data;

#### 3.3 Page Routing

Another approach to maintain consistency across regions. Since only the master database can accept write operations. This fact means we need to avoid serving pages that do database writes from slave region.

The load balancer has the capability to run in Layer 7 mode where it can examine the URI a user is requesting and make routing decisions based on that information. Load balancer can decide whether to send the request to front cluster in master region or slave region based on the page name and the user’s location. The most frequently accessed pages (home page, profiles, photo pages) don’t need to do any writes under normal operation.

Issue: When a user change something in his profile. The write request will be routed to master region. Data will be changed in database and will be deleted in cache in master region. When it is done, the user will read the page to check the update. The read request will be routed to slave region. Since the data hasn’t been replicated from master region to the slave region. The user will not see the update. How to fix it? We can work around this concern by setting a cookie. The load balancer checks the cookie and, if it notices that you wrote something in database in 30 seconds, it routes your read request to master region.

## 6.5. Cache replacement policies

LRU LFU

# 7. 存储设计

Sharding，Graph Database， SQL vs NoSQL

图数据库 Graph Databse（解决很多问题：类似共同好友： https://www.youtube.com/watch?v=41qdmKIIMz0）

# Appendix 其他技术点

## 1. 布隆过滤器

## 4. QPS相关

* QPS = 100 用你的笔记本做 Web 服务器就好了
* QPS = 1k 用一台好点的 Web 服务器就差不多了，需要考虑 Single Point Failure
* QPS = 1m需要建设一个1000台 Web 服务器的集群， 需要考虑如何 Maintenance（某一台挂了怎么办）

QPS和 Web Server (服务器) / Database (数据库) 之间的关系

* + - 一台 Web Server 约承受量是 1k 的 QPS

（考虑到逻辑处理时间以及数据库查询的瓶颈）

* + - 一台 SQL Database 约承受量是 1k 的 QPS

（如果 JOIN 和 INDEX query比较多的话，这个值会更小）

* + - 一台 NoSQL Database (Cassandra) 约承受量是 10k 的 QPS
    - 一台 NoSQL Database (Memcached) 约承受量是 1M 的 QPS

5. 分布式系统中的幂等性

<https://www.jianshu.com/p/475589f5cd7b>

<https://tech.meituan.com/distributed_system_mutually_exclusive_idempotence_cerberus_gtis.html>

数据的对象和范围你要考虑你的幂等的全局性：空间全局性和时间全局性。 空间全局性：比如是交易流水幂等还是用户ID幂等。是某种类型交易流水幂等，还是某个人|机构|渠道的交易流水幂等 时间全局性：是幂等几秒，还是几分钟，还是永远。不同的要求，可以有不一样的解决方案、难度和成本。幂等方案 对时间全局性要求高的，可能就必须选择DB这种持久化方案比较可靠，但是性能不够好啊（然后就要考虑loadmemory，以及数据同步的问题，就一步还要考虑实时性要求了） 在空间的要求中，根据不同的幂等范围，可以考虑分布式数据库（分布式集群全局流水号幂等）。还是某种少量数据幂等（可能只需要单台，做好主备）。

首先你要知道，分布式高并发并不意味着每个request都处理的很快，也不意味着机器之间就不共享数据了。其次，你可以把你所有带有副作用的task都给一个guid，最后写进数据库之前查一下这个guid是否已经被执行过了。

一个幂等的操作典型如：  
把编号为5的记录的A字段设置为0

这种操作不管执行多少次都是幂等的。

一个非幂等的操作典型如：  
把编号为5的记录的A字段增加1

这种操作显然就不是幂等的。

要做到幂等性，从接口设计上来说不设计任何非幂等的操作即可。  
譬如说需求是：  
当用户点击赞同时，将答案的赞同数量+1。  
改为：  
当用户点击赞同时，确保答案赞同表中存在一条记录，用户、答案。  
赞同数量由答案赞同表统计出来。