SCALE FROM ZERO TO MILLIONS OF USERS

Step 1 - With the growth of the user base, one single application server cannot handle the traffic anymore. We put the application server and the database server into two separate servers.

SCALE FROM ZERO TO MILLIONS OF USERS (cont)

Step 2 - The business continues to grow, and a single application server is no longer enough. So we deploy a cluster of application servers.

By **woshiamiaojiang**

cheatography.com/woshiamiaojiang/

Not published yet.
Last updated 19th September, 2023.

SCALE FROM ZERO TO MILLIONS OF USERS (cont)

Step 3 - Now the incoming requests have to be routed to multiple application servers, how can we ensure each application server gets an even load? The load balancer handles this nicely.

SCALE FROM ZERO TO MILLIONS OF USERS (cont)

Step 4 - With the business continuing to grow, the database might become the bottleneck. To mitigate this, we separate reads and writes in a way that frequent read queries go to read replicas. With this setup, the throughput for the database writes can be greatly increased.

SCALE FROM ZERO TO MILLIONS OF USERS (cont)

Step 5 - Suppose the business continues to grow. One single database cannot handle the load on both the inventory table and user table. We have a few options:

By **woshiamiaojiang**

cheatography.com/woshiamiaojiang/

Not published yet.
Last updated 19th September, 2023.

SCALE FROM ZERO TO MILLIONS OF USERS (cont)

1. Vertical scaling. Adding more power (CPU, RAM, etc.) to the database server. It has a hard limit.

SCALE FROM ZERO TO MILLIONS OF USERS (cont)

2. Horizontal scaling by adding more database servers.

SCALE FROM ZERO TO MILLIONS OF USERS (cont)

3. Adding a caching layer to offload read requests.

By **woshiamiaojiang**

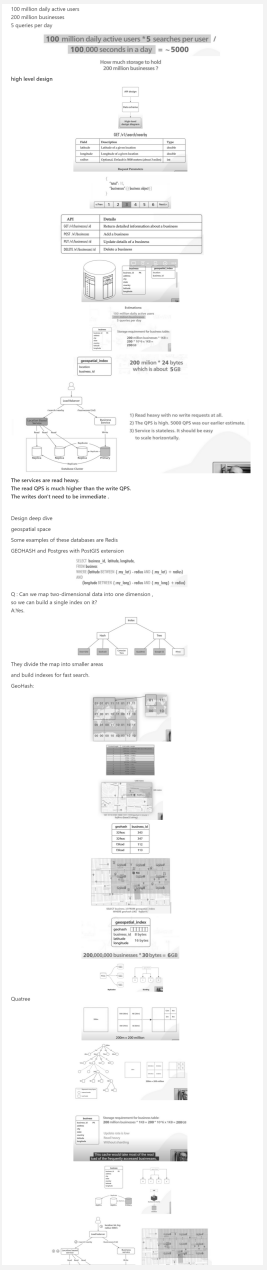
cheatography.com/woshiamiaojiang/

Not published yet.
Last updated 19th September, 2023.

SCALE FROM ZERO TO MILLIONS OF USERS (cont)

Step 6 - Now we can modularize the functions into different services. The architecture becomes service-oriented / microservice.

yelp



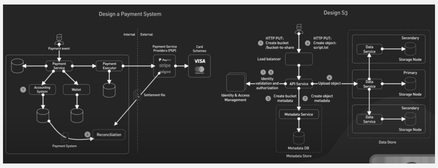
CHAPTER 4: DESIGN A RATE LIMITER

By **woshiamiaojiang**

cheatography.com/woshiamiaojiang/

Not published yet.
Last updated 19th September, 2023.
Page 4 of 100.

design S3



DESIGN A URL SHORTENER

- 每秒写操作数:1亿/ 24 /3600 = 1160
- 读操作:假设读操作与写操作的比例为10:1，读操作每秒操作数:1160 * 10 = 11,600
- 假设网址缩短服务将运行10年，这意味着我们必须支持1亿* 365 * 10 = 3650亿条记录。
- 假设平均URL长度为100。
- 10年以上存储需求:3650亿* 100字节* 10年= 365tb

步骤2 -提出高层次的设计并获得支持

在本节中，我们将讨论API端点、URL重定向和URL缩短流。

API端点

API端点促进了客户机和服务器之间的通信。我们将设计api restful。如果您不熟悉restful API，可以参考外部资料，例如参考资料\[1\]中的一个。URL缩短器主要需要两个API端点。

1. 网址缩短。要创建一个新的短URL，客户机发送一个POST请求，其中包含
一个参数:原始的长URL。API是这样的:
POST api / v1 /data/shorten
    - 请求参数:{longUrl: longURLString}
    - 返回shortURL
2. URL重定向。为了将短URL重定向到相应的长URL，客户端发送一个GET请求。API是这样的:
GET api / v1 /shortUrl
    - 返回longURL用于HTTP重定向

URL重定向

如figure8-1所示，在浏览器中输入一个tinyurl。一旦服务器接收到一个tinyurl请求，它将短URL更改为301重定向的长URL。

301重定向。 301重定向显示所请求的URL被"永久"移动到长URL。由于它是永久重定向的，因此浏览器会缓存响应，并且对同一URL的后续请求将不会发送到URL缩短服务。相反，请求被直接重定向到长URL服务器。

302重定向。 302重定向意味着URL被"暂时"移动到长URL，这意味着对同一URL的后续请求将被发送到URL缩短服务第一。然后，它们被重定向到长URL服务器。

每种重定向方法都有其优缺点。如果优先考虑的是减少服务器负载，使用301重定向是有意义的，因为只有相同URL的第一个请求被发送到URL缩短服务器。然而，如果分析是重要的，302重定向是一个更好的选择更容易跟踪点击率和点击来源。

哈希+碰撞分辨率

为了缩短长URL，我们应该实现一个散列函数，将长URL散列到7-字符串。一个直接的解决方案是使用众所周知的哈希函数，比如CRC32，MD5或SHA-1。

即使是最短的哈希值(来自CRC32)也太长(大于7)字符)。我们怎样才能缩短它?第一种方法是收集哈希值的前7个字符;然而，这种方法可能导致哈希冲突。为了解决哈希冲突，我们可以递归地附加一个new预定义的字符串，直到没有发现更多的冲突。

一般建议使用用非加密型哈希算法比如 MurmurHash 。因为，相比于 MD5，SHA 等加密型哈希算法，非加密型哈希算法往往效率更高!

我们拿 MurmurHash 来说，MurmurHash 当前最新的版本是 MurmurHash3，它能够产生出 32-bit 或 128-bit 哈希值。对于绝大部分场景来说，32-bit 的一般就已经够用了。

生成的哈希值是 10 进制的，为了缩短它的长度，我们可以将其转变为 62 进制即可。10 进制的 3394174629 转换为 62 进制就是 3HHBS5。

62进制转换

基转换是URL缩短器常用的另一种方法。基地转换帮助在不同的数字表示系统之间转换相同的数字。基地使用62转换，因为hashValue有62个可能的字符。

两种方法的比较

哈希+碰撞分辨率	62进制转换
固定短URL长度	短的URL长度不固定，他和ID一起上升
它不需要固定的ID生成器	此选项依赖于唯一的ID生成器
碰撞是可能的，必须解决	冲突是不可能的，因为ID是唯一的
不可能找出下一个可用的的短url,因为它不依赖于ID	如果新条目ID增加1，则很容易找出下一个可用的短的URL，这可能是一个安全问题

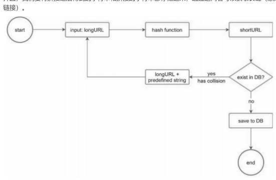
如何判断是否发生了哈希冲突呢?

不过，为了提高性能以及应对高并发，还是建议利用布隆过滤器解决这个问题。

如何解决哈希冲突呢?

解决办法其实也很简单。**如果发生哈希冲突，我们就在长链后拼接一个随机字符串。如果拼接了随机字符串还是发生哈希冲突那就再拼接一个随机字符串。**

并且，我们要将拼接之后得到的字符串和拼接的字符串都存储起来，通过这两者可以获取长链（原始链接）。



一个长链对应一个短链还是多个短链呢?

这个还是要看具体的业务需求。**个人建议是一个长链可以在不同的条件（比如生成短链的用户不同）下对应上不同的短链。这样的话，我们可以更好地对短链进行相关分析。**

举个例子：通过小码短连接后台，我们可以看到短连接的访问次数、访问人数等信息。

短链存储

如果我们使用 MySQL，PostgreSQL 这类关系型数据库存储的话，表结构大概是下面这样：

```
CREATE TABLE `url_map` (
`id` int(11) unsigned NOT NULL AUTO_INCREMENT,
`long_url` varchar(160) DEFAULT NULL COMMENT '长链',
`short_url` varchar(10) DEFAULT NULL COMMENT '短链',
PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Press 'space' for AI, '/' for commands...

步骤4 -打包

在本章中，我们讨论了API设计，数据模型，哈希函数，URL缩短，和URL重定向。

如果在面试结束时有额外的时间，这里有一些额外的谈话要点。

- 速率限制器:我们可能面临的一个潜在安全问题是，恶意用户发送数据包大量的URL缩短请求。速率限制器有助于过滤基于IP地址或其他过滤规则的请求。如果你想回忆一下关于速率限制，请参见"第四章:设计速率限制器"。
- Web服务器扩展:由于Web层是无状态的，所以很容易扩展Web层添加或删除web服务器。

typehead

- 快速响应时间。当用户输入一个搜索查询时，自动完成的建议必须足够快地显示出来。
足够快。一篇关于Facebook的自动完成系统的文章[1]显示，该系统需要在100毫秒内返回结果。
系统需要在100毫秒内返回结果。否则会造成卡顿。
- 相关性。自动完成的建议应该与搜索词相关。
- 排序。系统返回的结果必须按照流行度或其他排名模式进行排序。
模式。
- 可扩展性。系统可以处理高流量。
- 高可用性。当系统的一部分脱机、速度减慢或遇到意外的网络错误时，系统应保持可用和可访问性。
系统脱机、速度减慢或出现意外的网络错误时，系统应保持可用和可访问。

封底估计

- 假设1000万日活跃用户（DAU）。
- 一个人平均每天进行10次搜索。
- 每个查询字符串有20个字节的数据。
    - 假设我们使用ASCII字符编码。1个字符=1个字节
    - 假设一个查询包含4个词，每个词平均包含5个字符。
    - 那就是每个查询有4 x 5 = 20字节。
- 对于在搜索框中输入的每一个字符，客户都会向后端发送一个请求
以获得自动完成的建议。平均来说，每个搜索查询有20个请求被发送。例如
例如，当你完成输入时，以下6个请求被发送到后端
"晚餐"。
- ~24,000次/秒的查询（QPS）= 10,000,000用户*10次/天*20个字符/人
24小时/3600秒。
- 峰值QPS = QPS * 2 = ~48,000
- 假设每天的查询中有20%是新的。1000万*10次查询/天*20字节/每次
查询 * 20% = 0.4 GB。这意味着每天有0.4GB的新数据被添加到存储中。

第2步--提出高层次的设计并获得认同

- 数据收集服务。它收集用户的输入查询，并实时汇总它们。
实时处理对于大型数据集来说是不现实的；但是，这是一个很好的起点。
启点。我们将在深入研究中探索一个更现实的解决方案。
- 查询服务。给定一个搜索查询或前缀，返回5个最常搜索的术语。

数据收集服务

让我们用一个简化的例子来看看数据收集服务是如何工作的。假设我们有一个
频表，存储查询字符串和它的频率，如图13-2所示。在开始时
开始时，频率表是空的。后来，用户输入查询 "twitch"、"twitter"。
"twitter "和 "twillo "的顺序。图13-2显示了频率表的更新情况。

查询服务

- 查询：它存储查询字符串。
- 频率：它表示一个查询被搜索的次数。

第3步--设计深挖

- Trie数据结构
- 数据收集服务
- 查询服务
- 缩放存储
- Trie操作

- trie是一个树状的数据结构。
- 根代表一个空字符串。
- 每个节点存储一个字符，有26个子节点，每个子节点代表一个可能的字符。为了
为了节省空间，我们不画空链接。
- 每个树节点代表一个单词或一个前缀字符串。

获得前k个搜索次数最多的查询的步骤如下。

1. 找到前缀。时间复杂度。O(p)。
2. 从前缀节点遍历子树，得到所有有效的子节点。一个子节点是有效的，如果它
可以形成一个有效的查询字符串。时间复杂度。O(c)
3. 对子代进行排序，得到前k名。O(clogc)

在最坏的情况下，我们需要遍历整个 trie 来获得前 k 个结果。下面是两个优化方案。

1. 限制前缀的最大长度
2. 缓存每个节点的顶级搜索查询
让我们逐一来看看这些优化措施。

限制前缀的最大长度

用户很少在搜索框中输入一个很长的搜索查询。因此，可以说p是一个小的
整数，例如50。如果我们限制前缀的长度，"查找前缀 "的时间复杂度可以从O
前缀 "的时间复杂度可以从O(p)降低到O(小常数)，也就是O(1)。

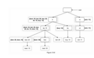
缓存每个节点的顶级搜索查询

为了避免遍历整个三角形，我们在每个节点存储最常用的前k个查询。
由于5到10个自动完成的建议对用户来说已经足够了，所以k是一个相对较小的数字。
在我们的具体案例中，只有前5个搜索查询被缓存。
通过在每个节点上缓存顶级搜索查询，我们大大降低了检索前5个查询的时间复杂性。
检索前5个查询的时间复杂性。然而，这种设计需要大量的空间来存储每个节点上的顶级查询。
每个节点都需要大量的空间。用空间换取时间是非常值得的，因为快速响应时间非常重要。
图13-8显示了更新后的 trie数据结构。前5个查询被存储在每个节点上。例如
例如，前缀为 "be "的节点存储以下内容。[best: 35, bet: 29, bee: 20, be: 15,
啤酒。10].

让我们重新审视一下应用这两个优化后的算法的时间复杂性。

1. 寻找前缀节点。时间复杂度。O(1)
2. 返回top k。由于top k的查询被缓存了，这一步的时间复杂度是O(1)。

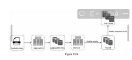
由于每个步骤的时间复杂度都降低到O(1)，我们的算法只需要
O(1)来获取前k个查询。



数据收集服务

在我们之前的设计中，每当用户键入一个搜索查询，数据就会实时更新。
由于以下两个原因，这种方法并不实用。

- 用户每天可能会输入数十亿次的查询。在每次查询中更新 trie
显著降低了查询服务的速度。
- 一旦建立了 trie，顶级建议可能不会有太大变化。因此，没有必要
频繁更新 trie。



Trie操作

Trie是自动完成系统的一个核心组件。让我们来看看Trie操作是如何
(创建、更新和删除)的工作。

创建

Trie是由工人使用聚合的数据创建的。数据的来源是来自分析
日志/数据库。

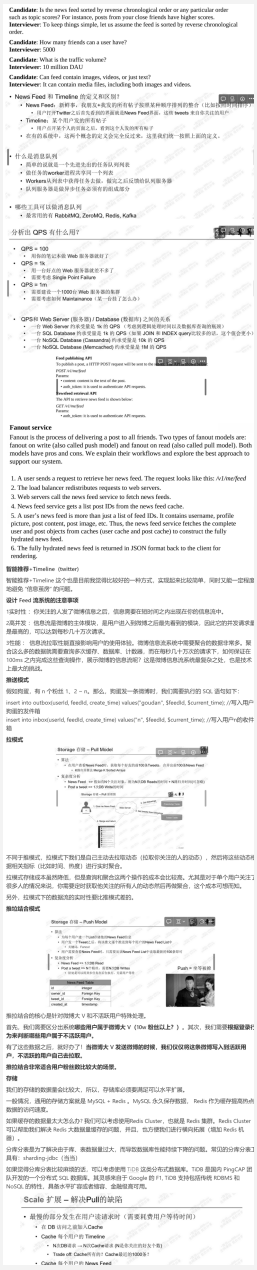
更新

有两种方法来更新 trie。
选项1：每周更新 trie。一旦一个新的 trie 被创建，新的 trie 将取代旧的
一个。
选项2：直接更新单个 trie 节点。我们尽量避免这种操作，因为它是
缓慢。然而，如果 trie 的大小很小，这是一个可以接受的解决方案。当我们更新一个
的时候，它的祖先直到根都必须被更新，因为祖先存储了子节点的顶级
查询的子节点。图13-13显示了一个更新操作的例子。在
左边，搜索查询 "啤酒 "的原始值是10。在右边，它被更新为
为30。正如你所看到的，该节点和它的祖先的 "啤酒 "值被更新为30。

删除

我们必须删除仇恨的、暴力的、色情的或危险的自动完成建议。
建议。我们在Trie Cache前面添加一个过滤层（图13-14）来过滤掉
不需要的建议。有了过滤层，我们就可以根据不同的过滤规则灵活地删除结果。

News Feed



Candidate: Is the news feed sorted by reverse chronological order or any particular order such as topic scores? For instance, posts from your close friends have higher scores.
Interviewer: To keep things simple, let us assume the feed is sorted by reverse chronological order.

Candidate: How many friends can a user have?
Interviewer: 5000

Candidate: What is the traffic volume?
Interviewer: 10 million DAU

Candidate: Can feed contain images, videos, or just text?
Interviewer: It can contain media files, including both images and videos.

- **News Feed** 和 **Timeline** 的定义和区别？
  - **News Feed**：新鲜事，我发生+朋友们的所有帖子按照某种顺序排列的整合（比如Facebook的News Feed）
  - 用户打开**Twitter**之后首先看到的界面就是**News Feed**界面，这即 **tweets** 来自你关注的用户
- **Timeline**：某个用户发的所有帖子
  - 用户点开某个人的首页之后，看到这个人发的所有帖子
  - 在你的系统中，这两个概念的定义会完全反过来，这里我们统一按照上面的定义。

- 什么是消息队列
  - 简单的说就是一个先进先出的任务队列列表
  - 设计当务之**worker**监控这个队列，一个死去
  - **Workers**从列表中获得任务去做，做完之后反馈给队列服务器
  - 任何服务器也能异步任务获得有的成果分

- 哪些主流可以做消息队列
  - 最常用的有 **RabbitMQ**, **ZeroMQ**, **Redis**, **Kafka**

分析出 **QPS** 有什么用？

- QPS = 100
  - 你做的笔记本和 Web 服务器就好了
- QPS = 1k
  - 找一台好点的 Web 服务器就差不多了
  - 需要考虑 **Single Point Failure**
- QPS = 1m
  - 需要建立一个1000台 Web 服务器的集群
  - 需要考虑如何 **Maintainance**（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** 约未定是处 为 05 QPS（考虑到资料处理时间以及数据库查询的瓶颈）
  - 一台 **SQL Database** 的未定是处 为 01 QPS（如果 JOIN 和 INDEX query比较多的话，这个值会更小）
  - 一台 **NoSQL Database (Cassandra)** 的未定是处 为 1k 到 QPS
  - 一台 **NoSQL Database (Memcached)** 的未定是处 为 1M 到 QPS

Fanout service

Fanout is the process of delivering a post to all friends. Two types of fanout models are: fanout on write (also called push model) and fanout on read (also called pull model). Both models have pros and cons. We explain their workflows and explore the best approach to support our system.

1. A user sends a request to retrieve her news feed. The request looks like this: */v1/me/feed*
2. The load balancer redistributes requests to web servers.
3. Web servers call the news feed service to fetch news feeds.
4. News feed service gets a list post IDs from the news feed cache.
5. A user's news feed is more than just a list of feed IDs. It contains username, profile picture, post content, post image, etc. Thus, the news feed service fetches the complete user and post objects from caches (user cache and post cache) to construct the fully hydrated news feed.
6. The fully hydrated news feed is returned in JSON format back to the client for rendering.

智能推荐 - Timeline（twitter）

智能推荐 + Timeline 这个也是目前我觉得比较好的一种方式，实现起来比较简单，同时又能一定程度地避免"信息爆炸"的问题。

设计 Feed 流系统的注意事项

1实时性：你关注的人发了微博信息之后，信息需要在较短的时间之内出现在你的信息流中。

2高并发：信息流是微博的主体模块，是用户进入到微博之后最先看到的模块，因此它的并发请求量是最高的，可以达到每秒几十万次请求。

3性能：信息流拉取性能直接影响用户的使用体验，微博信息流系统中需要聚合的数据非常多，聚合这么多的数据如果都查询多次缓存，数据库，计数器，即使每秒几十万次的请求下，如何将延迟100ms 之内完成这些查询操作，展示微博的信息流呢？这是微博信息流系统最复杂之处，也是技术上最大的挑战。

推送模式

假如我是，有 n 个粉丝 1，2 ~ n，那么，则是发一条微博时，我们需要执行的 SQL 语句如下：

```
insert into outbox(userid, feedid, create_time) values("goudan", $feedid, $current_time); //写入用户的发件箱
insert into inbox(userid, feedid, create_time) values("n", $feedid, $current_time); //写入用户n的收件箱
```

拉模式

不同于推模式，拉模式下我们是自己主动去找写动态（拉取你关注的人的动态），然后将这些动态根据相关指标（比如时间，热度）进行实时聚合。

拉模式存储成本是线性的，但是查询时耗会随着个插件的成本会比较高，尤其是对于单个用户关注了很多人的情况来说，你需要实时获取他关注的所有人的动态然后再做聚合，这个成本可想而知。

另外，拉模式下的数据高度的实时性都比推模式差距。

推拉结合模式

推拉结合的核心是针对特殊大 V 和不活跃用户特殊处理。

首先，我们需要区分出你感觉**哪些用户属于普通大 V（10w 粉丝以上？）**，其次，我们需要**根据登录行为来判断哪些用户属于不活跃用户。**

有了这些数据之后，就好办了！**当普通大 V 发送的时候的时候，我们仅仅将这条微博写入到活跃用户、不活跃的用户自己会访问。**

推拉结合非常适合用户粉丝数比较大的场景。

存储

我们的存储的数据量会比较大，所以，存储集必须要满足可以水平扩展。

一般情况，选择现存储方案就是 MySQL + Redis，MySQL 永久保存数据，Redis 作为缓存提高热点数据的访问速度。

如果缓存的数据量太大怎么办？我们可以考虑使用Redis Cluster，也就是 Redis 集群。Redis Cluster 可以用来很好解决 Redis 大数据量缓存的问题，并且，也方便我们进行横向扩展（就加 Redis 机器）。

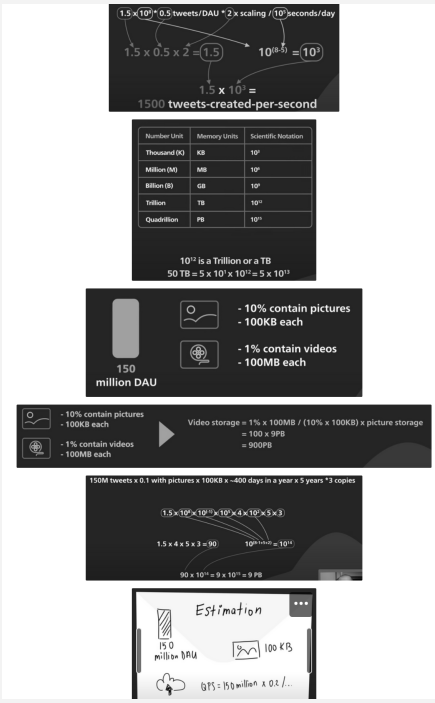
分库分表是为了解决由于库，表数据量过大，而导致数据操作维护等下降的问题，常见的分库分表工具有：sharding-jdbc（当当）

如果觉得分库分表比较麻烦的话，可以考虑使用 TiDB 这类分布式数据库。TiDB 是国内 PingCAP 团队开发的一个分布式 SQL 数据库。其灵感来自于 Google 的 F1, TiDB 支持包括传统 RDBMS 和 NoSQL 的特性，具备水平扩容或者缩容，金融级高可用。

Scale 扩展 – 解决Pull的缺陷

- 最慢的部分发生在用户请求刷时（需要耗费用户等待时间）
  - 在 DB 的网上前加入**Cache**
  - Cache 每个用户的 **Timeline**
    - N次DB请求 --> N次Cache请求 (N次命中的有较大个数)
    - Trade-off: Cache存储个 Cache最近的1000条？
  - Cache 每个用户的 **News Feed**

BACK-OF-THE-ENVELOPE ESTIMATION

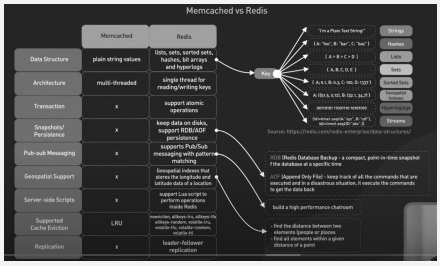


DAU

QPS

STORAGE

Memcached vs Redis



设计youtube

- 假设该产品有500万日活跃用户（DAU）。
- 用户每天观看5个视频。
- 10%的用户每天上传1个视频。
- 假设平均视频大小为300MB。
- 每天需要的总存储空间。500万*10%*300MB=150TB
- CDN费用。
  - 当云CDN提供视频服务时，你要为从CDN传输出来的数据付费。
    CDN。
  - 让我们使用亚马逊的CDN CloudFront进行成本估算（图14-2）[3]。假设100%的流量来自美国。每GB的平均成本为0.02美元。
    为简单起见，我们只计算视频流的成本。
  - 500万1个视频0.3GB*0.02美元=每天150,000美元。

第2步--提出高层次的设计并获得认同

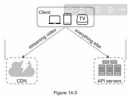


Figure 14-3

视频上传

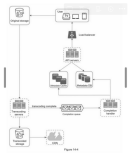


Figure 14-4

流程a：上传实际视频

Flow b: update the metadata

视频流流程

当我们设计一个视频流媒体服务时，我们必须选择正确的流媒体协议来支持我们的用例。

第3步--设计深挖

视频转码的重要性在于以下原因。

- 原始视频消耗大量的存储空间。一个小时的高清晰度视频以每秒60帧的速度录制，可以占用几百GB的空间。
- 许多设备和浏览器只支持某些类型的视频格式。因此，将视频编码为不同的格式是很重要的。因此，出于兼容性的考虑，将视频编码为不同的格式很重要。
- 为了确保用户观看高质量的视频，同时保持流畅的播放，这是一个好主意。
- 网络条件可以改变，特别是在移动设备上。为了确保视频顺畅

有向无环图（DAG）模型

下面是一些
可以应用于视频文件的一些任务。

- 检查。确保视频有良好的质量，没有畸形。
- 视频编码。视频被转换以支持不同的分辨率、编解码、比特率。
  等。图14-9显示了一个视频编码文件的例子。
- 缩略图。缩略图可以由用户上传或由系统自动生成。
  系统自动生成。
- 水印。覆盖在你的视频上面的图像包含关于你的视频的识别信息
  关于你的视频。

视频转码架构

该架构有六个主要部分：预处理器、DAG调度器、资源管理器、
任务工作者，临时存储，以及作为输出的编码视频。

预处理器有4个职责。

1. 视频拆分。视频流被分割成进一步分割成更小的图片组（GOP）。
   对齐。GOP是一组/大块的被按特定顺序排列。每个块是一个
   可独立播放的单元，通常为几秒钟的长度。
2. 一些旧的移动设备或浏览器可能不支持视频分割。预处理程序通过GOP对齐方式为老客户分割
   预处理器通过GOP对齐的方式为老客户分割视频。
3. 3.DAG生成。处理器根据客户的配置文件生成DAG
4. 缓存数据。预处理器是分段视频的缓存器。为了提高可靠性
   预处理器将GOP和元数据存储在临时存储器中。如果视频编码失败
   系统可以使用持久化的数据进行重试操作。

资源管理员

- 任务队列。它是一个优先级队列，包含要执行的任务。
- 工作队列。它是一个优先级队列，包含工人的利用率信息。
- 运行队列。它包含关于当前正在运行的任务和正在运行的工人的信息
  任务的信息。
- 任务调度器。它挑选最佳的任务/工作者，并指示所选的任务工作者来
  执行该任务。

任务调度器

- 任务调度器从任务队列中获得最高优先级的任务。
- 任务调度器从工人队列中获得运行该任务的最佳任务工人。
- 任务调度员指示选择的任务工作者运行任务。
- 任务调度员将定任务/工作者信息并将其放入运行队列。
- 一旦任务完成，任务调度员就会将任务从运行队列中移除。

系统优化

速度优化：视频上传的并行化

将一个视频作为一个整体上传是低效的。我们可以通过以下方式将视频分割成更小的块状物
GOP对齐，如图14-22所示。

速度优化：将上传中心放在靠近用户的地方

另一种提高上传速度的方法是在全球范围内建立多个上传中心（图14-24）。
图14-24）。在美国的人可以把视频上传到北美的
上传中心，而中国的人可以把视频上传到亚洲的上传中心。为了实现
这一点，我们使用CDN作为上传中心。

chat system

- 支持多种设备。同一账户可同时登录多个客户。
  同时登录多个客户。
- 推送通知

第二步--提出高水平的设计并获得认同

模拟一个由服务器交互的简单：轮询、长轮询和WebSocket。

高级别的设计

无状态服务。有状态服务和第三方集成

客户端与聊天服务器保持一个持久的WebSocket连接。用于实时消息传递。

- 聊天服务器促进消息的发送/接收。
- 存在管理服务器管理在线/离线状态。
- API服务器处理一切，包括用户登录、注册、更改资料等。
- 通知服务器发送推送通知。
- 最后，键值存储用于存储聊天历史。当一个离线的用户时，她会看到她以前所有的聊天记录。

存储

需要键值NS存储数据的功能，如登录、查看您的联络人内容、新闻资料您的
消息等。这些模拟应该由数据密码写架来支持。

- 对于1对1的聊天应用程序，读与写的比例约为1:1，
  选择正确的存储系统，支持我们所有的用例是非关重要的。我们
  我们推荐键值存储，理由如下：
- 键值存储允许轻松地进行横向扩展。
- 键值存储为访问数据提供了非常低的延迟。
- 关系型数据库不擅长对较长型长期访问数据。当索引增长
  大时，随机访问就很昂贵。
- 键值存储被其他接近用可靠的聊天应用所采用。比如说
  Facebook的messenger和Discord都使用键值存储。Facebook创建变用HBase[4]，而Discord使用
  Cassandra[5]。

数据模型

数据模型

假定，我们决定了使用键值存储作为我们的存储层。最重要的
数据查询数据。让我们开始看一下。
1对1聊天的消息表
图13-3显示了1对1聊天的消息表。主键是message_id，它
帮助决定消息的顺序。我们不能依靠created_at来决定消息的顺序
因为两条消息可能同时被创建。

Figure 13-10

群组聊天的消息表
图13-10显示了群聊的消息表。显示主键是
(channel_id, message_id)，channel和group会话主要代表相同的意思。channel_id
是分区键，因为群聊中的所有查询都在一个频道中操作。

Figure 13-11

Message ID
如何生成message_id是一个值得探索的有趣话题。Message_id和负责
负责确保消息的顺序。为了确定消息的顺序，
message_id必须满足以下两个要求。

- ID必须是唯一的。
- ID应该是可以按时间排序的，也就是说，新行的ID比旧行的ID高。

第二个方法是使用一个全局性的64位序列号生成器，如Snowflake[6]。

最后一种方法是使用本地序列号生成器。本地意味着ID只在在一个给定组唯一的。本地ID工作的原因
是，在一对一的会话或一个较内强特消息序列在一对一的会话或一个较的组内有就足够了。这种方法
更容易与全局ID的实现相比，这种方法更容易实现。

第3步--设计深挖

服务发现 ZK

1. 用户A试图登录到应用程序。
2. 负载均衡器将登录请求发送到API服务器。
3. 后台验证用户后，服务发现为用户A找到最佳聊天服务器。
   在这个例子中，服务器2被选中。服务器信息被返回给用户A。
4. 用户A通过WebSocket连接到聊天服务器2。

消息流

1. 用户A向聊天服务器1发送了一条聊天信息。
2. 聊天服务器1从ID生成器获得一个消息ID。
3. 聊天服务器1将消息发送到消息同步队列中。
4. 消息被存储在一个键值存储器中。
   a. 如果用户B在线，该消息被转发到用户B所在的聊天服务器2。
   连接。
   b. 如果用户B是离线的，则从推送通知（PN）服务器发送新送通知。
5. 聊天服务器2将消息转发给用户B。
   用户B和聊天服务器2之间有一个持久的WebSocket连接。

多个设备和跨设备同步

每个设备都有一个名为cur_max_message_id的变量。它记录了设备上最新的
的变量。它记录了设备上最新的消息ID。满足以下两个条件的消息被认为是
作为新信息。

- 收件人ID等于当前登录的用户ID。
- 键值存储中的资源ID大于cur_max_message_id。
  在每个设备上有不同的cur_max_message_id。消息同步很容易，因为每个
  设备可以从键值存储中获得自己消息。

小群组聊天流程

首先，来自用户
首先，用户A的消息被复制到每个组成的消息同步队列中：一个给用户B，另一个给用户C。
你可以把消息同步队列想象成收件人的收件箱。这种设计
这种设计对小型群组的未来应用的好处选择，因为

- 已简化了消息同步流程，因为每个客户只需要检查自己的收件箱就可以得到
  新消息。
- 当群组人数较少时，在每个接收者的收件箱中存储一份副本不会太昂贵。
  原因。

在线提示

当服务器负责管理在线状态，并通过WebSocket与客户进行通信。
用户登录、注销、断线

在线状态扇出

用户A的朋友是如何知道状态变化的？ 图12-19解释了它是如何工作的。
存在服务器使用一个发布-订阅模型，其中每个朋友对独自一个
通道。当用户A的在线状态改变时，它将事件发布到三个频道。
频道A、B、C，和A-D。这三个频道相用户B、C和D订阅。
上述设计对小型群的用户群很有效的。例如，微信使用了一个类似的
因为它的用户群的上限是500人。

假设一个群组有100,000个成员。每一个状态变化将产生100,000个事件。为了解决这个性能瓶颈。一
个可能的解决方案是一个可能的解决方案是只在用户进入群组或手动刷新时获取在线状态。

第4步 - 总结

- 扩展聊天应用程序以支持媒体文件，如照片和视频。媒体文件是

DESIGN A WEB CRAWLER

Step 1 - Understand the problem and establish design scope

Back of the envelope estimation
The following estimations are based on many assumptions, and it is important to communicate with the interviewer to be on the same page.
• Assume 1 billion web pages are downloaded every month.
• QPS: 1,000,000,000 / 30 days / 24 hours / 3600 seconds = ~400 pages per second.
• Peak QPS = 2 * QPS = 800
• Assume the average web page size is 500k.
• 1-billion-page x 500k = 500 TB storage per month. If you are unclear about digital storage units, go through "Power of 2" section in Chapter 2 again.
• Assuming data are stored for five years, 500 TB * 12 months * 5 years = 30 PB. A 30 PB storage is needed to store five-year content.

Step 2 - Propose high-level design and get buy-in

Figure 9-4

Step 1: Add seed URLs to the URL Frontier
Step 2: HTML Downloader fetches a list of URLs from URL Frontier.
Step 3: HTML Downloader gets IP addresses of URLs from DNS resolver and starts downloading.
Step 4: Content Parser parses HTML pages and checks if pages are malformed.
Step 5: After content is parsed and validated, it is passed to the "Content Seen?" component.
Step 6: "Content Seen" component checks if a HTML page is already in the storage.
• If it is in the storage, this means the same content in a different URL has already been processed. In this case, the HTML page is discarded.
• If it is not in the storage, the system has not processed the same content before. The content is passed to Link Extractor.
Step 7: Link extractor extracts links from HTML pages.
Step 8: Extracted links are passed to the URL filter.
Step 9: After links are filtered, they are passed to the "URL Seen?" component.
Step 10: "URL Seen" component checks if a URL is already in the storage, if yes, it is processed before, and nothing needs to be done.
Step 11: If a URL has not been processed before, it is added to the URL Frontier.

Step 3 - Design deep dive

Up until now, we have discussed the high-level design. Next, we will discuss the most important building components and techniques in depth:
• Depth-first search (DFS) vs Breadth-first search (BFS)

DFS is usually not a good choice because the depth of DFS can be very deep.
BFS is commonly used by web crawlers and is implemented by a first-in-first-out (FIFO) queue. In a FIFO queue, URLs are dequeued in the order they are enqueued. However, this implementation has two problems:
1Most links from the same web page are linked back to the same host.

2Standard BFS does not take the priority of a URL into consideration.

• URL frontier

URL frontier helps to address these problems. A URL frontier is a data structure that stores URLs to be downloaded.

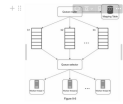


Figure 9-5

• 89% guess it. 1st is to Each give crawlers URLs from its own host.
• Queue selector: Each worker thread is mapped to a FIFO queue, and it only downloads URLs from that queue. The queue selection logic is done by the Queue selector.
• Worker thread 1 to N. A worker thread downloads web pages one by one from the same host. A delay can be added between two download tasks.

Figure 9-7

• Prioritizer. It takes URLs as input and computes the priorities.
• Queue 1 to to the. Each queue has an assigned priority. Queues with high priority are selected more frequently.
• Queue selector. Randomly choose a queue with a bias towards queues with higher priority.

Figure 9-8 presents the URL frontier design, and it contains two modules:
• Front queues: manage prioritization
• Back queues: manage politeness

Freshness
Few strategies to optimize freshness are listed as follows:
• Recrawl based on web pages' update history.
• Prioritize URLs and recrawl important pages first and more frequently.

Storage for URL Frontier
We adopted a hybrid approach. The majority of URLs are stored on disk, so the storage space is not a problem. To reduce the cost of reading from the disk and writing to the disk, we maintain buffers in memory for enqueue/dequeue operations. Data in the buffer is periodically written to the disk.
• HTML Downloader

Performance optimization
Below is a list of performance optimizations for HTML downloader.

1.Distributed crawl

2. Cache DNS Resolver Our DNS cache keeps the domain name to IP address mapping and is updated periodically by cron jobs.

3. Locality

4. Short timeout To avoid long wait time, a maximal wait time is specified.
• Robustness

We present a few approaches to improve the system robustness:
• Consistent hashing: This helps to distribute loads among downloaders. A new downloader server can be added or removed using consistent hashing.
• Save crawl states and data: To guard against failures, crawl states and data are written to a stor system. A disrupted crawl can be restarted easily by loading saved states and data.
• Exception handling: Errors are inevitable and common in a large-scale system. The crawler mus handle exceptions gracefully without crashing the system.
• Data validation: This is an important measure to prevent system errors.

• Extensibility

1PNG Downloader module is plugged-in to download PNG files.
2Web Monitor module is added to monitor the web and prevent copyright and trademark infringements.

• Detect and avoid problematic content

This section discusses the detection and prevention of redundant, meaningless, or harmful content.

1. Redundant content
   As discussed previously, nearly 30% of the web pages are duplicates. Hashes or checksums help to detect duplication [11].
2. Spider traps
   A spider trap is a web page that causes a crawler in an infinite loop.
3. Data noise
   Some of the contents have little or no value, such as advertisements, code snippets, spam URLs, etc. Those contents are not useful for crawlers and should be excluded if possible.

Step 4 - Wrap up
In this chapter, we first discussed the characteristics of a good crawler: scalability, politeness, extensibility, and robustness. Then, we proposed a design and discussed key components. Finally, we discussed some optimizations and data analytics. Building a crawler is not a trivial task and full of traps. Even though we have covered many topics, we still miss many relevant talking points:
• Server-side rendering: Numerous websites use scripts like JavaScript, AJAX, etc to

DESIGN A UNIQUE ID GENERATOR

步骤1 -理解问题并明确设计范围

要求如下:

- id必须唯一。
- id仅为数值。
- id适合64位。
- id按日期排序。
- 每秒生成超过10,000个唯一id的能力

步骤2 -提出高层次的设计并获得支持

- 多主复制

  这种方法使用数据库的auto_increment特性。而不是增加下一个ID每增加1，我们增加k，其中k是正在使用的数据库服务器的数量。下一个生成的ID等于同一台服务器上一个生成的ID加2。这解决了一些可伸缩性问题，因为id可以随着数据库服务器的数量而扩展。然而，这种策略有一些**主要的缺点**:

  难以扩展多个数据中心
  在多个服务器上，id不会随时间增长。
  当添加或删除服务器时，它不能很好地扩展

- 通用唯一标识符(UUID)

  优点:
  - 生成UUID很简单。服务器之间不需要协调，所以不会有出现任何同步问题。
  - 系统易于扩展，因为每个web服务器负责生成id他们消费。ID生成器可以很容易地扩展与web服务器。

  缺点:
  - id长度为128位，但我们的要求是64位。
  - id不随时间增长。
  - id可以是非数字的。

- 票据服务器

  优点:
  - 数字id。
  - 易于实现，适用于中小型应用程序。

  缺点:
  - 单点故障。单票证服务器意味着如果票证服务器宕机。所有依赖于它的系统将面临问题。为了避免单点故障，我们可以设置多个票证服务器。然而，这将带来新的挑战，如数据同步。

- Twitter雪花策略

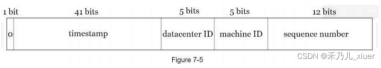


Figure 7-5

CSDN @禾乃儿_xiuer

- List
  - 符号位:1位。它总是0。这是为将来使用而保留的。它可能是用于区分有符号数和无符号数。
  - 时间戳:41位。自epoch或自定义epoch以来的毫秒数。我们使用Twitter默认纪元1288834974657，相当于Nov 04, 2010, 01:42:54 UTC。
  - 数据中心ID: 5位，即2 ^ 5 = 32个数据中心。
  - 机器ID: 5位，每个数据中心有2 ^ 5 = 32台机器。
  - 序列号:12位。对于在该机器/进程上生成的每个IDNumber加1。该数字每毫秒重置为0。

步骤3 -设计深度潜水

可以用41位表示的最大时间戳是

2 ^ 41 - 1 = 2199023255551毫秒(ms)，这给了我们~ 69年=2199023255551毫秒/ 1000秒/ 365天/ 24小时/ 3600秒。这意味着ID生成器将工作69年，并具有接近今天日期延迟的自定义历元时间溢出时间。69年后，我们将需要一个新的纪元时间或采用其他技术迁移id。

序列号

序列号是12位，这给了我们2 ^ 12 = 4096个组合。这个字段是0，除非同一服务器在一毫秒内生成多个ID。理论上，机器可以每毫秒最多支持4096个新id。

步骤4 -打包

在本章中，我们讨论了设计唯一ID生成器的不同方法:多主复制、UUID、票据服务器Twitter雪花状唯一ID生成器。我们选择snowflake，因为它支持我们所有的用例，并且在分布式环境中可扩展环境。

如果在面试结束时有额外的时间，这里有一些额外的谈话要点:

- 时钟同步。在我们的设计中，我们假设ID生成服务器具有相同的功能时钟。当服务器在多个核心上运行时，这个假设可能不成立。时同样的挑战也存在于多机器场景中。时钟同步的解决方案有超出本书范围的.然而，了解问题的存在是很重要的。网络时间协议是解决这个问题的最流行的解决方案。
- 节长度调整。例如，序列号更少，但时间戳位更多对于低并发性和长期应用程序是有效的。
- 高可用性。由于ID生成器是一个任务关键型系统，因此必须高度可用。
  恭喜你走了这么远!现在给自己点鼓励吧。好工作!

DESIGN CONSISTENT HASHING

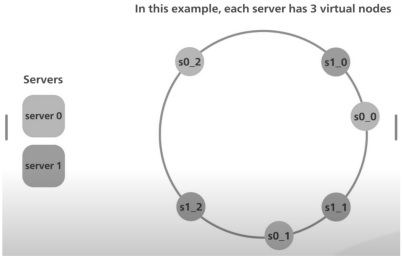
一致性哈希算法步骤如下:

1. 定义一个哈希空间，通常是一个32位或64位的整数范围。
2. 将每个节点通过哈希函数计算其标识（如IP地址或名称）的哈希值，并将其映射到环上的位置。
3. 将每个数据通过哈希函数计算其标识（如键）的哈希值，并将其映射到环上的位置。
4. 当需要存储或查找数据时，通过哈希函数计算数据的标识的哈希值，并在环上找到离该哈希值最近的节点。
5. 将数据存储在该节点上或从该节点获取数据。

```
serverIndex = hash(key) % N,
where N is the size of the server pool.
```

基本方法有两个问题
一致性哈希算法是由Karger等人在MIT[1]提出的。基本步骤是:

使用统一分布的哈希函数将服务器和密钥映射到环上。
要找出一个键映射到哪个服务器，从键位置顺时针走，直到找到环上的第一个服务器。
这种方法存在两个问题。首先，不可能保持相同的规模考虑到可以添加或删除服务器的所有服务器在环上的分区。一个分区是相邻服务器之间的哈希空间。有可能是分配给每个服务器的环上的分区要么非常小，要么相当大。在figure5-10中，如果s1被删除后，s2的分区(用双向箭头突出显示)是s0的两倍大s3的分区。



Amazon DynamoDB Apache Cassandra	Data partitioning
Content delivery networks	Distribute web contents evenly
Load balancers	Distribute persistent connections evenly

一致性哈希环（Consistent Hash Ring）是一种用于分布式系统中数据分片和负载均衡的算法。它解决了传统哈希算法在节点增减时导致数据迁移的问题，并提供了高效的数据查找和负载均衡能力。

在一致性哈希环中，节点和数据都被映射到一个固定大小的环上。环的范围是一个固定的哈希空间，通常是一个32位或64位的整数。每个节点通过计算其标识（如IP地址或名称）的哈希值来确定在环上的位置。

当需要存储或查找数据时，首先将数据的标识（如键）计算哈希值，并在环上找到离该哈希值最近的节点。然后，将数据存储在该节点上或从该节点获取数据。这样，每个节点负责一部分哈希空间，数据在节点之间分布均匀。

虚拟节点（Virtual Nodes）是一种扩展一致性哈希环的技术。它通过引入多个虚拟节点来增加节点的数量，从而提高数据的均匀性。每个真实节点可以映射到多个虚拟节点，这些虚拟节点在环上均匀分布。这样，即使节点数量较少，数据仍然可以在虚拟节点上均匀分布，从而提高负载均衡性能。

从节点（Replica Nodes）是一种用于提高数据冗余和容错性的技术。在一致性哈希环中，每个数据通常只由一个节点负责存储和处理。为了增加数据的冗余和可靠性，可以为每个节点创建多个从节点。从节点是主节点的副本，它们存储相同的数据，并可以在主节点不可用时接管服务。从节点通过复制主节点的数据来保持数据的一致性。

GOSSIP通信是一种用于分布式系统中节点之间通信的协议。它基于充分信息传播（Epidemic Information Dissemination）的思想。通过节点之间的相互交流和信息传递来实现数据的同步和协调。

在GOSSIP通信中，每个节点都维护一个关于其他节点的信息列表。节点定期选择一些其他节点进行通信，并将自己的信息传播给这些节点。被选择的节点也会将自己的信息传播给其他节点，从而实现信息的扩散和传播。

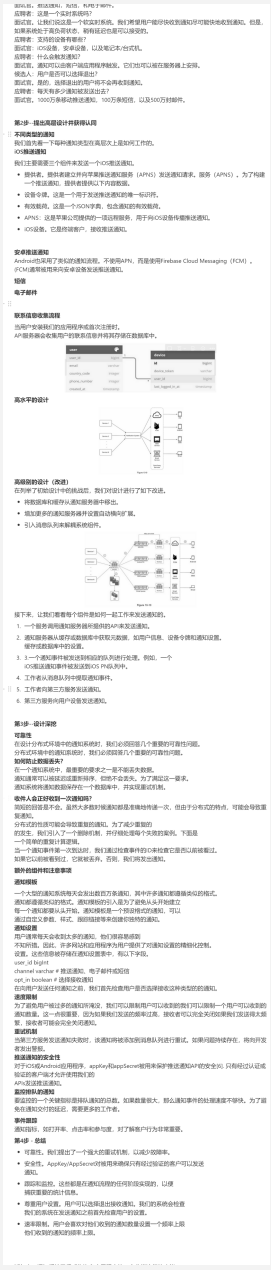
通过GOSSIP通信，节点可以相互了解彼此的状态和数据，并进行数据的同步和协调。当节点之间存在数据不一致或冲突时，GOSSIP通信可以帮助节点进行数据的合并和解决冲突，从而保持系统的一致性。

GOSSIP通信具有以下特点:

- **去中心化**：每个节点都可以主动选择其他节点进行通信，没有中心节点的依赖。这种去中心化的通信方式使得系统更加灵活和可扩展。
- **容错性**：由于节点之间相互交流和信息传递，即使部分节点出现故障或不可用，系统仍然可以通过其他节点进行信息传播和数据同步，从而提高系统的容错性。
- **自适应性**：GOSSIP通信可以根据系统的负载和网络状况进行自适应调整。节点可以根据需要选择通信的节点数量和频率，以适应不同的环境和需求。
- **随机性**：节点选择通信的目标节点通常是随机选择的，这样可以避免节点之间的同步和通信过于集中，提高系统的负载均衡性能。

总结起来，GOSSIP通信是一种用于分布式系统中节点之间通信的协议，通过节点之间的相互交流和信息传递来实现数据的同步和协调。它具有去中心化、容错性、自适应性和随机性等特点，可以帮助构建可靠和高效的分布式系统。

设计一个通知系统



DESIGN GOOGLE DRIVE

初步估计

- 假设该应用程序有5000万注册用户和1000万DAU。
- 用户得到10GB的免费空间。
- 假设用户每天上传2个文件。平均文件大小为500KB。
- 读写比为1:1。
- 分配的总空间。5000万*10GB=500Petabyte
- 上传API的QPS：1000万*2次上传/24小时/3600秒=~240
- 峰值QPS = QPS * 2 = 480

我们主要需要3个API：上传文件，下载文件，以及获取文件的修订。

下面是一个可恢复上传API的例子。
https://api.example.com/files/upload?uploadType=resumable
参数：

- uploadType=resumable
- 数据。要上传的本地文件。

可恢复上传是通过以下3个步骤实现的[2]。

- 发送初始请求以检索可恢复的URL。
- 上传数据并监控上传状态。
- 如果上传受到干扰，恢复上传。

2. 从Google Drive下载一个文件
   示例API：https://api.example.com/files/download
   参数：

- path: 下载文件的路径。
  例子参数。
  {
  "path": "/recipes/soup/best_soup.txt"
  }

3. 获取文件修订版
   示例API：https://api.example.com/files/list_revisions
   参数：
   - path。你想获得修订历史的文件的路径。
   - limit：要返回的最大修订数。

   示例参数：
   {
   "path": "/recipes/soup/best_soup.txt",
   "limit": 20
   }

研究你可以改进的地方。以下是你发现的几个方面。

- 负载平衡器。添加一个负载平衡器来分配网络流量。负载平衡器可以确保均匀分布的流量，如果一个网络服务器发生故障，它将重新分配流量。
- 网络服务器。在添加了一个负载平衡器后，更多的网络服务器可以很容易地被添加/删除。根据流量负载，可以很容易地添加/删除更多的网络服务器。
- 元数据数据库。将数据库移出服务器以避免单点故障。同时，设置数据复制和分片，以满足可用性和可扩展性要求。可扩展性要求。
- 文件存储。Amazon S3用于文件存储。为了确保可用性和耐久性。文件被复制在两个独立的地理区域。

同步冲突

我们的系统将同一文件的两个副本
同一文件的两个副本：用户2的本地副本和服务器上的最新版本（图15-9）。用户2可以选择合并这两个文件，或者用另一个版本覆盖一个版本。

高层设计

用户： 用户通过浏览器或移动应用程序使用该应用程序。
区块服务器。 区块服务器将区块上传到云存储。块存储，被称为块级存储，是一种在基于云的环境中存储数据文件的技术。一个文件可以被分割成几个块，每个块都有一个独特的哈希值，存储在我们的元数据数据库中。每个区块都被视为一个独立的对象，并存储在我们的存储系统中(S3). 为了重建一个文件，块以特定的顺序被连接。至于块的大小，我们使用Dropbox作为参考：它将一个块的最大尺寸设定为4MB[6]。
云存储。 一个文件被分割成更小的块并存储在云存储中。
冷存储。 冷存储是一种为存储非活动数据而设计的计算机系统，意味着文件长期不被访问。
负载平衡器。 负载平衡器在API服务器之间均匀地分配请求。
API服务器。 这些服务器负责除上传流程以外的几乎所有工作。API服务器用于用户认证、管理用户资料、更新文件元数据等。
元数据数据库。 它存储用户、文件、块、版本等的元数据。请注意文件存储在云中，元数据数据库包含元数据。
元数据缓存。 一些元数据被缓存起来，以便快速检索。
通知服务。 它是一个发布者/订阅者系统，允许数据在某些事件发生时从通知服务传输到客户端。当某些事件发生时，从通知服务传输到客户端。在我们的具体案例中，通知当一个文件在其他地方被添加/编辑/删除时，通知服务会通知相关客户，这样他们就可以拉出最新的变化。
离线备份队列。 如果客户端处于离线状态，无法提取最新的文件变化，那么离线的备份队列存储信息，因此当客户在线时，变化将被同步。

第3步--设计深究

- Delta同步。当一个文件被修改时，只有被修改的块被同步，而不是使用同步算法的整个文件。使用同步算法[7][8]。
- 压缩。对块进行压缩可以大大减少数据大小。因此，根据文件类型，使用压缩算法对块进行压缩。例如例如，gzip和bzip2被用来压缩文本文件。不同的压缩算法需要不同的压缩算法来压缩图像和视频。
- 一个文件被分割成更小的块。
- 每个区块都使用压缩算法进行压缩。
- 为了确保安全，每个区块在被发送到云存储之前都要进行加密。
- 块被上传到云存储。

高一致性要求

ACID属性必须以编程方式
纳入同步逻辑中。在我们的设计中，我们选择了关系型数据库，因为ACID是原生支持的。

元数据数据库

Figure 15-13

上传流程

让我们讨论一下客户上传文件时发生了什么。为了更好地理解这个流程，我们绘制如图15-14所示的序列图。

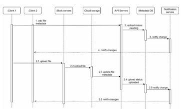
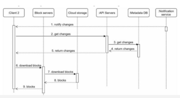


Figure 15-14

在图15-14中，两个请求被并行发送：添加文件元数据和上传文件到云存储。这两个请求都来自客户端1。

下载流程



DESIGN A KEY-VALUE STORE

- 键值对大小较小，一般小于10kb。
- 具备大数据存储能力。
- 高可用性:即使在故障期间，系统也能快速响应。
- 高可扩展性:系统可扩展以支持大型数据集。
- 自动扩展:服务器的添加/删除应该基于流量自动。
- 可调一致性。
- 低延迟

单服务器键值存储

开发驻留在单个服务器中的键值存储很容易。一个直观的方法是在哈希表中存储键值对，哈希表将所有内容保存在内存中。尽管内存访问速度很快，由于空间的原因，可能无法将所有内容装入内存约束。为了在单个服务器上容纳更多数据，可以进行两种优化:

- 数据压缩
- 仅将频繁使用的数据存储在内存中，其余数据存储在磁盘上

系统组件

在本节中，我们将讨论用于构建web服务的以下核心组件和技术键值存储:

- 数据分区：一致性哈希+虚拟节点
- 数据复制：将键映射到哈希环上的某个位置后，从该位置顺时针走并选择环上的前N个服务器来存储数据副本。
- 一致性：如果W + R > N，则保证强一致性，因为至少有一个数据最新的重叠节点，保证数据的一致性。

当R = 1, W = N时，系统优化为快速读取。

如果W = 1, R = N，则系统优化为快速写入。

如果W + R > N，则保证强一致性(通常N = 3, W = R = 2)。

如果W + R <= N，则不能保证强一致性。

- 不一致解决**Inconsistency resolution: versioning. vector clock**
- 故障处理:gossip
  - 每个节点维护一个节点成员列表，包含成员id和心跳计数器。
  - 各节点定时增加心跳计数器。
  - 每个节点周期性地向一组随机节点发送心跳，这些随机节点依次发送心跳传播到另一组节点。
  - 一旦节点接收到心跳，成员列表将更新为最新信息。
  - 如果心跳没有增加超过预定义的时间，成员是被视为离线。
- **处理临时故障** 由于s2在不可用读取和写入将暂时由s3处理。当s2重新上线，s3也会将数据交还给s2。
- **处理永久性故障**: 比较副本上的每个数据块并进行更新每个副本到最新版本。Merkle树用于不一致检测和尽量减少传输的数据量。
- **处理数据中心中断**: 异地多活
- 系统架构图

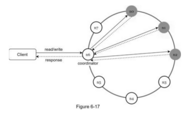


Figure 6-17

- 写路径

1. 写请求保存在提交日志文件中。
2. 数据保存在内存缓存中。
3. .当内存缓存已满或达到预定义阈值时，数据将刷新到磁盘上的SSTable[9]。注意:排序字符串表(SSTable)是一个<key, value>的排序列表对。

- 读路径

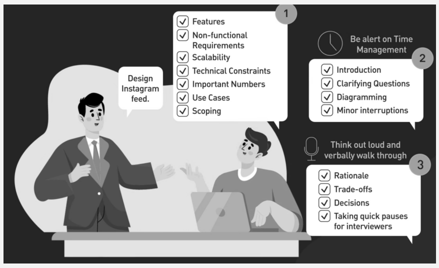
1. 系统首先检查数据是否在内存中。否= >步骤2。
2. 如果数据不在内存中，系统将检查布隆过滤器。
3. 布隆过滤器用于找出哪些sstable可能包含密钥。
4. sstable返回数据集的结果。
5. 数据集的结果返回给客户端。

总结

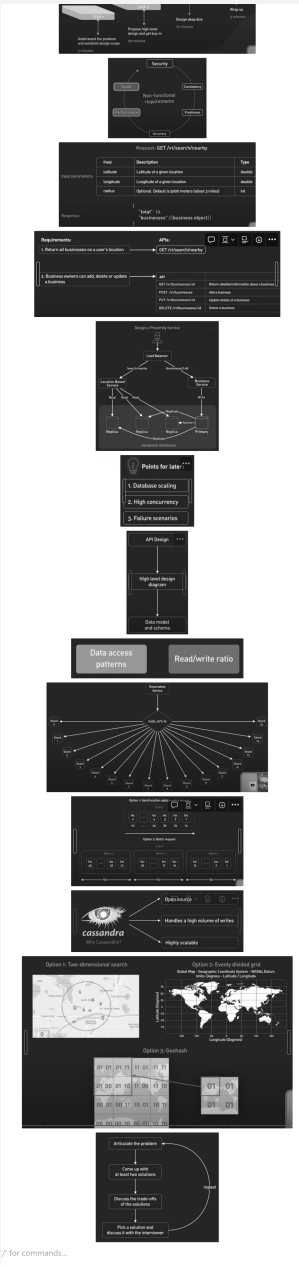
本章涵盖了许多概念和技术。为了刷新你的记忆，下面是表总结了分布式键值的特性和相应的技术商店。

目标/问题	技术
具备存储大数据	使用一些散列在服务器之间分散负载
高可用性读取	数据复制，多数据中心设置
高可用写入	使用矢量时钟进行版本控制和冲突解决
数据分区	一致性哈希
增量可伸缩	一致性哈希
非均质	一致性哈希
可调的一致性	群体共识Quorum consensus
处理临时故障	草率的法定人数和暗示交接
处理永久性故障	merkle树
处理数据中心中断	跨数据中心复制

How to Crack Any System Design Interview



A FRAMEWORK FOR SYSTEM DESIGN INTERVIEWS



estimation: dau, qps, storage
API
high level design diagram
data model and schema
articulate the problem

how to prepare system design

https://www.yuque.com/snailclimb/mf2-z3k/hizgws

Live Streaming Platform Work