

系统设计访谈:分步指南

许多软件工程师在系统设计面试 (SDI) 中遇到困难,主要是因为以下三个原因

原因:

- SDI 的非结构化性质,要求他们进行开放式设计
没有标准答案的问题。
- 他们缺乏开发大型系统的经验。 · 他们没有为SDI 做好准备。

与编码面试一样,没有刻意准备 SDI 的候选人大多表现不佳,尤其是在 Google、Facebook、亚马逊、微软等顶级公司。

在这些公司中,表现不高于平均水平的候选人获得报价的机会有限。另一方面,良好的表现总是会带来更好的工作机会(更高的职位和薪水),因为它显示了候选人处理复杂系统的能力。

在本课程中,我们将遵循逐步的方法来解决多个设计问题。首先,让我们完成以下步骤:

第 1 步:需求澄清

询问我们正在解决的问题的确切范围总是一个好主意。设计问题大多是开放式的,并且没有一个正确的答案,这就是为什么在面试初期澄清歧义变得至关重要。花足够时间来定义系统最终目标的候选人总是有更好的机会在面试中取得成功。另外,由于我们只有 35-40 分钟的时间来设计一个(据称)大型系统,因此我们应该明确我们将重点关注系统的哪些部分。

让我们通过设计类似 Twitter 的服务的实际示例来扩展这一点。以下是设计 Twitter 时应回答的一些问题,然后再继续下一步:

- 我们服务的用户是否能够发布推文并关注其他人? · 我们是否还应该设计创建和显示用户的时间线?
- 推文中是否包含照片和视频? · 我们只关注后端还是也开发前端? · 用户能够搜索推文吗?
- 我们是否需要显示热门话题? · 新的(或重要的)推文会有推送通知吗?

所有这些问题将决定我们的最终设计会是什么样子。

第二步:系统接口定义

定义系统期望提供哪些 API。这不仅会建立预期的确切合同

从系统中,但也会确保我们没有得到任何错误的要求。我们类似 Twitter 的服务的一些示例如下:

```
postTweet(user_id, tweet_data, tweet_location, user_location, timestamp, ...) generateTimeline(user_id,  
current_time, user_location, ...) markTweetFavorite(user_id, tweet_id, timestamp,  
...)
```

步骤 3:粗略估计

估计我们要设计的系统的规模总是一个好主意。当我们稍后关注扩展、分区、负载平衡和缓存时,这也会有所帮助。

- 系统的预期规模是多少（例如,新推文数量、推文浏览量、每秒生成的时间线数量等）？ · 我们需要多少存储空间?如果用户的推文中可以包含照片和视频,我们将得到不同的数字。
- 我们期望使用多少网络带宽?这对于决定我们如何进行至关重要
管理流量并平衡服务器之间的负载。

步骤 4:定义数据模型

尽早定义数据模型将阐明数据如何在系统的不同组件之间流动。稍后,它将指导数据分区和管理。考生应该能够识别系统的各种实体,它们如何相互交互,以及数据管理的不同方面,如存储、传输、加密等。以下是我们类似 Twitter 服务的一些实体:

用户:用户 ID、姓名、电子邮件、DoB、CreationData、LastLogin 等。

推文: TweetID、内容、TweetLocation、NumberOfLikes、TimeStamp 等。

用户关注:用户 ID1、用户 ID2

最喜欢的推文:用户 ID、推文 ID、时间戳

我们应该使用哪种数据库系统? NoSQL 会像[Cassandra](#)一样吗最适合我们的需求,或者我们应该使用类似 MySQL 的解决方案?我们应该使用什么样的块存储来存储照片和视频?

第五步:高层设计

绘制一个框图,其中 5-6 个方框代表我们系统的核心组件。我们应该确定端到端解决实际问题所需的足够组件。

对于 Twitter,在较高层面上,我们需要多个应用程序服务器来服务所有读/写请求,并在它们前面提供负载均衡器以进行流量分配。如果我们假设我们将有更多的读取流量（与写入相比）,我们可以决定使用单独的服务器来处理这些情况。在后端,我们需要一个高效的数据库来存储所有的推文,并且可以

支持海量读取。我们还需要一个分布式文件存储系统来存储照片和视频。

第六步 :详细设计

深入挖掘两个或三个组件;面试官的反馈应该始终指导我们系统的哪些部分需要进一步讨论。我们应该能够展示不同的方法及其优缺点,并解释为什么我们更喜欢一种方法而不是另一种方法。请记住,没有单一的答案,唯一重要的是考虑不同选项之间的权衡,同时牢记系统约束。

- 由于我们将存储大量数据,我们应该如何对数据进行分区
分发到多个数据库?我们是否应该尝试将用户的所有数据存储在同一个数据库中?它可能会导致什么问题?
- 我们将如何处理经常发推文或关注很多人的热门用户? · 由于用户的时间线
将包含最新(且相关)的推文,我们是否应该尝试以针对扫描最新推文进行优化的方式存储数据? · 我们应该在多大程度上以及在哪一层引入缓存来加快速度? · 哪些组件需要更好的负载平衡?

第 7 步 :识别并解决瓶颈

尝试讨论尽可能多的瓶颈以及缓解瓶颈的不同方法。

- 我们的系统中是否存在单点故障?我们正在采取什么措施来缓解这种情况? · 我们是否有足够的数据副本,以便即使我们失去一些服务器,我们仍然可以为我们的服务提供服务。
用户?
- 同样,我们是否有足够的运行不同服务的副本,以便少数故障不会发生?
不会导致整个系统关闭吗?
- 我们如何监控我们的服务绩效?我们是否会在关键时刻收到警报
组件出现故障或性能下降?

概括

简而言之,面试过程中的准备和组织是系统设计面试成功的关键。上述步骤应指导您在设计系统时保持正轨并涵盖所有不同方面。

让我们应用上述指南来设计 SDI 中要求的一些系统。

设计一个 URL 缩短服务,例如 小网址

让我们设计一个像 TinyURL 这样的 URL 缩短服务。该服务将提供重定向到长 URL 的短别名。类似服务 :bit.ly、goo.gl、qlink.me 等 难度级别:简单

1.为什么需要URL缩短?

URL 缩短用于为长 URL 创建较短的别名。我们将这些缩短的别名称为“短链接”。当用户点击这些短链接时,他们会被重定向到原始 URL。短链接可以节省很多显示、打印、发送消息或发布推文时的空格。此外,用户不太可能错误输入较短的 URL。

例如,如果我们通过 TinyURL 缩短这个页面:

https://www.eduvative.io/collection/page/5668639101419520/5649050225344512/5668600_916475904/

我们会得到:

<http://tinyurl.com/jlg8zpc>

缩短的 URL 大小几乎是实际 URL 的三分之一。

URL 缩短用于跨设备优化链接、跟踪单个链接以分析受众和营销活动绩效以及隐藏关联的原始 URL。

如果您还没有使用过tinyurl.com在此之前,请尝试创建一个新的缩短的 URL,并花一些时间浏览他们的服务提供的各种选项。这将对你理解本章有很大帮助。

2. 系统的要求和目标

您应该始终在面试开始时澄清要求。一定要问清楚问题来找出面试官心目中系统的确切范围。

我们的URL缩短系统应满足以下要求：

功能要求：

1. 给定一个 URL,我们的服务应该生成一个更短且唯一的别名。这就是所谓的短关联。
2. 当用户访问短链接时,我们的服务应将其重定向到原始链接。
3. 用户应该可以选择为其 URL 选择自定义短链接。
4. 链接将在标准默认时间跨度后过期。用户应该能够指定过期时间。

非功能性要求：

1. 系统应该是高可用的。这是必需的,因为如果我们的服务出现故障,所有 URL 重定向将开始失败。
2. URL 重定向应该实时发生,延迟最小。
3. 缩短的链接不应该是可猜测的（不可预测的）。

扩展要求：

1. 分析;例如,重定向发生了多少次?
2. 我们的服务也应该可以由其他服务通过 REST API 访问。

3. 容量估计和约束

我们的系统读取量很大。与新的 URL 缩短相比,将会有大量的重定向请求。我们假设读写比率为 100:1。

流量估计:假设我们每月有 500M 新的 URL 缩短,读/写比为 100:1,我们预计同期会有 50B 的重定向:

$$100 * 500M \Rightarrow 50B$$

我们系统的每秒查询数 (QPS) 是多少?每秒新缩短的 URL:

$$5 \text{ 亿} / (30 \text{ 天} * 24 \text{ 小时} * 3600 \text{ 秒}) \approx 200 \text{ 个 URL/秒}$$

考虑到 100:1 的读/写比率,每秒 URL 重定向将为:

$$100 * 200 \text{ 个 URL/秒} = 20K/\text{秒}$$

存储估计:假设我们将每个 URL 缩短请求 (以及相关的缩短链接) 存储 5 年。由于我们预计每月会有 5 亿个新 URL, 因此我们预计存储的对象总数将为 300 亿个:

$$5\text{亿} \times 5\text{年} \times 12\text{个月} = 300\text{亿}$$

我们假设每个存储的对象大约有 500 字节 (只是一个大概的估计 我们稍后会深入研究)。我们需要 15TB 的总存储空间:

$$300\text{亿} \times 500\text{字节} = 15\text{TB}$$

带宽估计:对于写入请求,由于我们预计每秒有 200 个新 URL,因此我们服务的总传入数据将为每秒 100KB:

$$200 \times 500\text{字节} = 100\text{KB}/\text{秒}$$

对于读取请求,由于我们预计每秒约有 20K 个 URL 重定向,因此我们服务的总传出数据将为每秒 10MB:

$$20K \times 500\text{字节} = \sim 10\text{MB}/\text{s}$$

内存估算:如果我们要缓存一些经常访问的热点URL,需要多少

我们需要内存来存储它们吗?如果我们遵循 80-20 规则,即 20% 的 URL 产生 80% 的流量,我们希望缓存这 20% 的热门 URL。

由于我们每秒有 20K 个请求,因此我们每天将收到 17 亿个请求:

$$20K \times 3600\text{秒} \times 24\text{小时} = \sim 17\text{亿}$$

要缓存其中 20% 的请求,我们需要 170GB 内存。

$$0.2 \times 17\text{亿} \times 500\text{字节} = \sim 170\text{GB}$$

这里需要注意的一点是,由于会有很多重复请求 (同一 URL),因此,我们的实际内存使用量将小于 170GB。

高水平估计:假设每月有 5 亿个新 URL 和 100:1 的读:写比率,以下是我们服务的高水平估计的摘要:

新网址	200/秒
URL 重定向	20K/s
传入数据	100KB/秒
传出数据	10MB/秒
存储 5 年	15TB
缓存内存	170GB

4. 系统API

一旦我们最终确定了需求,定义系统 API 总是一个好主意。这
应明确说明系统的期望。

我们可以使用 SOAP 或 REST API 来公开我们服务的功能。以下是用于创建和删除 URL 的 API 的定义：

createURL(api_dev_key,original_url,custom_alias=无,user_name=无,expire_date=无)

参数：

api_dev_key (string): 注册账户的API开发者密钥。除其他外，这将用于根据分配的配额限制用户。

Original_url (字符串) :要缩短的原始 URL。

custom_alias (字符串) :URL 的可选自定义键。

user_name (字符串) :编码中使用的可选用户名。

expire_date (字符串) :缩短的 URL 的可选到期日期。

返回：(字符串)

成功插入返回缩短的 URL；否则，它返回一个错误代码。

删除URL (api_dev_key,url_key)

其中“url_key”是表示要检索的缩短 URL 的字符串。成功删除将返回“URL 已删除”。

我们如何发现并防止滥用行为？恶意用户可以通过消耗当前设计中的所有 URL 密钥来使我们破产。为了防止滥用，我们可以通过 api_dev_key 限制用户。每个 api_dev_key 可以限制为在某个时间段内一定数量的 URL 创建和重定向（可以将每个开发人员密钥设置为不同的持续时间）。

5. 数据库设计

在面试的早期阶段定义数据库模式将有助于理解数据

各个组件之间的流动，随后将指导数据分区。

关于我们将存储的数据的一些观察：

1. 我们需要存储数十亿条记录。
2. 我们存储的每个对象都很小（小于1K）。
3. 记录之间不存在任何关系，除了存储哪个用户创建了 URL 之外。
4. 我们的服务是重读的。

数据库架构：

我们需要两张表：一张用于存储有关 URL 映射的信息，一张用于存储用户的信息
创建短链接的数据。

网址		用户	
PK	哈希值:varchar(16)	PK	用户ID:int
原始URL:	varchar(512)	名称:	varchar(20)
创建日期:	日期时间	电子邮件:	varchar(32)
到期日期:	数据时间		创建日期:日期时间
[观众不支持]		上次登录:数据时间	

我们应该使用什么样的数据库?由于我们预计存储数十亿行,并且我们不需要使用对象之间的关系 - 像[DynamoDB](#)这样的[NoSQL 键值存储,卡桑德拉或里亚克](#)是一个更好的选择。 NoSQL 选择也更容易扩展。请参阅[SQL 与 NoSQL 为了](#)更多细节。

6. 基本系统设计和算法

我们这里要解决的问题是,如何为给定的 URL 生成一个简短且唯一的密钥。

在第 1 节的 TinyURL 示例中,缩短的 URL 是 “<http://tinyurl.com/jlg8zpc>” 。该 URL 的最后六个字符是我们要生成的短密钥。我们将在这里探索两种解决方案:

A。对实际 URL 进行编码

我们可以计算一个唯一的哈希值 (例如, [MD5](#)或[SHA256](#),等)给定的 URL。然后可以对散列进行编码以供显示。此编码可以是 base36 ([az,0-9]) 或 base62 ([AZ, az, 0-9]),如果我们添加 “-”和 “.”我们可以使用base64编码。一个合理的问题是,短密钥的长度应该是多少? 6、8 或 10 个字符。

使用 Base64 编码,6 个字母长的密钥将产生 $64^6 = \sim 687$ 亿个可能的字符串

使用 Base64 编码,8 个字母长的密钥将产生 $64^8 = \sim 281$ 万亿个可能的字符串

对于 68.7B 的唯一字符串,我们假设六个字母键足以满足我们的系统。

如果我们使用 MD5 算法作为哈希函数,它将产生一个 128 位的哈希值。经过base64编码后,我们将得到一个超过21个字符的字符串 (因为每个base64字符编码哈希值的6位)。由于每个短密钥只有 8 个字符的空间,我们将如何选择我们的密钥

然后?我们可以取前 6 个 (或 8 个)字母作为密钥。但这可能会导致密钥重复,因此我们可以从编码字符串中选择一些其他字符或交换一些字符。

我们的解决方案有哪些不同的问题?我们的编码方案存在以下几个问题:

- 1.如果多个用户输入相同的URL,他们可以获得相同的缩短的URL,这不是可以接受。

2. 如果 URL 的一部分是 URL 编码的怎么办?例如, <http://www.eduvative.io/distributed.php?id=设计>,和<http://www.eduvative.io/distributed.php%3Fid%3Ddesign>除了 URL 编码之外,其他都相同。

问题的解决方法:我们可以将递增的序列号附加到每个输入 URL 以使其唯一,然后生成它的哈希值。不过,我们不需要将此序列号存储在数据库中。这种方法可能出现的问题是序列号不断增加。

能溢出吗?附加递增的序列号也会影响服务的性能。

另一种解决方案是将用户 ID (应该是唯一的)附加到输入 URL。但是,如果用户尚未登录,我们将不得不要求用户选择唯一性密钥。即使在此之后,如果出现冲突,我们也必须继续生成密钥,直到获得唯一的密钥。

缩短 URL 的请求流程

1共 9 个

b. 离线生成密钥

我们可以拥有一个独立的密钥生成服务 (KGS),它预生成随机的六个字母字符串并将它们存储在数据库中(我们称之为密钥数据库)。每当我们想要缩短 URL 时,我们只需获取已生成的密钥之一并使用它即可。这种方法将使事情变得非常简单和快速。我们不仅不对 URL 进行编码,而且不必担心重复

或碰撞。KGS 将确保插入 key-DB 的所有密钥都是唯一的

并发会导致问题吗?密钥一旦使用,就应该在数据库中进行标记,以确保它不会被再次使用。如果有多个服务器同时读取密钥,我们可能会遇到两个或多个服务器尝试从数据库读取相同密钥的情况。我们如何解决这个并发问题呢?

服务器可以使用 KGS 读取/标记数据库中的密钥。KGS 可以使用两张表来存储密钥:一张用于尚未使用的密钥,一张用于所有已使用的密钥。一旦 KGS 向其中一台服务器提供密钥,它就可以将它们移动到已使用的密钥表中。KGS 可以始终将一些密钥保留在内存中,以便在服务器需要时可以快速提供它们。

为简单起见,一旦 KGS 将某些密钥加载到内存中,它就可以将它们移动到已使用的密钥表中。

这确保每个服务器获得唯一的密钥。如果 KGS 在将所有加载的密钥分配给某个服务器之前就死掉了,我们将浪费这些密钥 考虑到我们拥有大量的密钥,这是可以接受的。

KGS 还必须确保不要将相同的密钥提供给多个服务器。为此,它必须同步 (或锁定)持有密钥的数据结构,然后再从删除密钥并将其提供给服务器。

密钥数据库的大小是多少?通过base64编码,我们可以生成68.7B唯一的六个字母密钥。如果我们需要一个字节来存储一个字母数字字符,我们可以将所有这些键存储在:

$$6 \text{ (每个密钥的字符)} * 68.7\text{B} \text{ (唯一密钥)} = 412 \text{ GB}.$$

KGS 不是单点故障吗?是的。为了解决这个问题,我们可以拥有一个 KGS 的备用副本。每当主服务器挂掉时,备用服务器就可以接管并生成和提供密钥。

每个应用程序服务器可以缓存密钥数据库中的一些密钥吗?是的,这肯定可以加快速度。尽管在这种情况下,如果应用程序服务器在消耗所有密钥之前死亡,我们最终将丢失这些密钥。这是可以接受的,因为我们有 68B 独特的六字母键。

我们如何执行键查找?我们可以在数据库或键值存储中查找键来获取完整的 URL。如果存在,则向浏览器发出“HTTP 302 重定向”状态,并在请求的“位置”字段中传递存储的 URL。如果我们的系统中不存在该密钥,请发出“HTTP 404 Not Found”状态或将用户重定向回主页。

我们应该对自定义别名施加大小限制吗?我们的服务支持自定义别名。用户可以选择他们喜欢的任何“键”,但提供自定义别名不是强制性的。然而,对自定义别名施加大小限制以确保我们拥有一致的 URL 数据库是合理的(并且通常是可取的)。假设用户可以为每个客户密钥指定最多 16 个字符(如上面的数据库架构所示)。

URL 缩短的高级系统设计

7. 数据分区和复制

为了扩展我们的数据库,我们需要对其进行分区,以便它可以存储有关数十亿个 URL 的信息。我们需要提出一个分区方案,将数据划分并存储到不同的数据库服务器。

A。基于范围的分区 :我们可以根据 URL 的首字母或哈希键将 URL 存储在单独的分区中。因此,我们将所有以字母 “A”开头的 URL 保存在一个分区中,将那些以字母 “B”开头的 URL 保存在另一个分区中,依此类推。这种方法称为基于范围的分区。

我们甚至可以将某些不常出现的字母组合到一个数据库分区中。我们应该提出一个静态分区方案,以便我们始终可以以可预测的方式存储/查找文件方式。

这种方法的主要问题是它可能导致服务器不平衡。例如:我们决定将所有以字母 “E”开头的 URL 放入数据库分区中,但后来我们意识到以字母 “E”开头的 URL 太多了。

b.基于哈希的分区 :在此方案中,我们采用所存储对象的哈希值。然后我们根据哈希计算要使用哪个分区。在我们的例子中,我们可以使用“键”的哈希值或实际 URL 来确定存储数据对象的分区。

我们的哈希函数会将 URL 随机分布到不同的分区中 (例如,我们的哈希函数总是可以将任何键映射到 [1…256] 之间的数字) ,这个数字代表分区我们在其中存储我们的对象。

这种方法仍然会导致分区过载,这可以通过使用一致性哈希来解决。

8. 缓存

我们可以缓存经常访问的URL。我们可以使用一些现成的解决方案,例如 Memcache,它可以存储完整的 URL 及其各自的哈希值。应用程序服务器在访问后端存储之前,可以快速检查缓存中是否有所需的 URL。

我们应该有多少缓存?我们可以从每日流量的 20% 开始,根据客户的使用模式,我们可以调整我们需要的缓存服务器数量。根据上面的估计,我们需要 170GB 内存来缓存 20% 的日常流量。由于现代服务器可以拥有 256GB 内存,因此我们可以轻松地将所有缓存安装到一台机器中。或者,我们可以使用几个较小的服务器来存储所有这些热门 URL。

哪种缓存驱逐策略最适合我们的需求?当缓存已满,并且我们想用更新/更热门的 URL 替换链接时,我们会如何选择?最近最少使用 (LRU)对于我们的系统来说是一个合理的策略。根据此策略,我们首先丢弃最近最少使用的 URL。我们可以使用[连接哈希图](#)或者类似的数据结构来存储我们的 URL 和哈希值,它还将跟踪最近访问过的 URL。

为了进一步提高效率,我们可以复制缓存服务器以在它们之间分配负载。

如何更新每个缓存副本?每当出现缓存未命中时,我们的服务器就会访问后端数据库。每当发生这种情况时,我们都可以更新缓存并将新条目传递给所有缓存副本。每个副本都可以通过添加新条目来更新其缓存。如果副本已经具有该条目,则可以简单地忽略它。

访问缩短的 URL 的请求流程

1 个 (共 11 个)

9. 负载均衡器 (LB)

我们可以在系统中的三个位置添加负载均衡层：

- 1. 客户端和应用程序服务器之间
- 2. 应用程序服务器和数据库服务器之间
- 3. 应用程序服务器和缓存服务器之间

最初,我们可以使用简单的循环方法,在后端服务器之间平均分配传入请求。该LB实现简单,不会引入任何开销。这种方法的另一个好处是,如果服务器死机,LB 会将其从轮换中删除,并停止向其发送任何流量。

循环负载均衡的一个问题是没有考虑服务器负载。如果服务器过载或速度缓慢,负载均衡器不会停止向该服务器发送新请求。为了解决这个问题,可以采用更智能的负载均衡解决方案,定期查询后端服务器的负载情况,并据此调整流量。

10. 清除或数据库清理

条目应该永远保留还是应该被清除?如果达到了用户指定的过期时间,链接会发生什么情况?

如果我们选择主动搜索过期链接来删除它们,这会给我们的数据库带来很大的压力。相反,我们可以慢慢删除过期链接并进行惰性清理。我们的服务将确保只删除过期的链接,尽管有些过期的链接可以存在更长的时间,但永远不会返回给用户。

- 每当用户尝试访问过期链接时,我们可以删除该链接并向用户。
- 可以定期运行单独的清理服务,以从我们的存储和缓存中删除过期的链接。该服务应该非常轻量级,并且可以安排仅在预计用户流量较低时运行。
- 我们可以为每个链接设置一个默认的过期时间(例如,两年)。
- 删除过期链接后,我们可以将密钥放回到密钥数据库中以供重复使用。
- 我们是否应该删除一段时间内(例如六个月)未访问过的链接?这可能很棘手。由于存储变得越来越便宜,我们可以决定永远保留链接。

URL缩短的详细组件设计

11. 遥测

短 URL 使用了多少次,用户位置是什么,等等?我们如何存储这些

统计数据?如果它是在每个视图上更新的数据仓库的一部分,那么当流行的 URL 受到大量并发请求的冲击时会发生什么?

一些值得跟踪的统计数据:访问者所在的国家/地区、访问日期和时间、引用点击的网页、浏览器或访问页面的平台。

12. 安全和权限

用户能否创建私有 URL 或允许特定用户组访问 URL?

我们可以在数据库中存储每个 URL 的权限级别 (公共/私有)。我们还可以创建一个单独的表来存储有权查看特定 URL 的 UserID。如果用户没有权限并尝试访问 URL,我们可以发回错误 (HTTP 401)。鉴于我们正在存储

我们的数据存储在像 Cassandra 这样的 NoSQL 宽列数据库中,表存储权限的密钥将是“哈希”(或 KGS 生成的“密钥”)。这些列将存储有权查看 URL 的用户的 UserID。

设计 Pastebin

让我们设计一个类似 Pastebin 的 Web 服务,用户可以在其中存储纯文本。该服务的用户将进入一段文本并获取随机生成的 URL 来访问它。类似服务:pastebin.com、pasted.co、hopapp.com 难度级别:简单

1. Pastebin是什么？

Pastebin 之类的服务使用户能够通过网络（通常是互联网）存储纯文本或图像

并生成唯一的 URL 来访问上传的数据。此类服务还用于通过网络快速共享数据，因为用户只需传递 URL 即可让其他用户看到它。

如果您还没有使用过pastebin.com在此之前，请尝试在那里创建一个新的“粘贴”，并花一些时间浏览他们的服务提供的不同选项。这将对你理解本章有很大帮助。

2. 系统的要求和目标我们的 Pastebin 服务应

满足以下要求：

功能要求：

1. 用户应该能够上传或“粘贴”他们的数据并获得唯一的 URL 来访问它。
2. 用户将只能上传文本。
3. 数据和链接将在特定时间后自动失效；用户还应该能够指定过期时间。
4. 用户应该能够选择为其粘贴选择自定义别名。

非功能性要求：

1. 系统应高度可靠，上传的任何数据都不应丢失。
2. 系统应该是高可用的。这是必需的，因为如果我们的服务出现故障，用户将无法访问他们的粘贴。
3. 用户应该能够以最小的延迟实时访问他们的粘贴。
4. 粘贴链接不应是可猜测的（不可预测的）。

扩展要求：

1. 分析，例如，粘贴内容被访问了多少次？
2. 我们的服务也应该可以由其他服务通过 REST API 访问。

3. 一些设计考虑

Pastebin 与 URL 缩短服务共享一些要求，但我们还应该牢记一些额外的设计注意事项。

用户一次可以粘贴的文本量的限制应该是多少？我们可以限制用户粘贴的大小不得超过 10MB，以阻止滥用服务。

我们应该对自定义 URL 施加大小限制吗？由于我们的服务支持自定义 URL，因此用户可以选择他们喜欢的任何 URL，但提供自定义 URL 不是强制性的。然而，对自定义 URL 施加大小限制是合理的（而且通常是可以接受的），这样我们就有个一致的 URL 数据库。

4. 容量估计和约束

我们的服务将需要大量阅读;与新创建的粘贴相比,将会有更多的读取请求。我们可以假设读和写之间的比例为 5:1。

流量估计: Pastebin 服务预计不会有类似于 Twitter 或 Facebook 的流量,我们假设每天有 100 万个新粘贴添加到我们的系统中。这给我们留下了每天阅读 500 万次。

每秒新粘贴数:

$$1M / (24 \text{ 小时} * 3600 \text{ 秒}) \approx 12 \text{ 粘贴/秒}$$

粘贴每秒读取次数:

$$5M / (24 \text{ 小时} * 3600 \text{ 秒}) \approx 58 \text{ 次读取/秒}$$

存储预估: 用户最多可上传 10MB 数据;通常,类似 Pastebin 的服务用于共享源代码、配置或日志。此类文本并不大,因此我们假设每个粘贴平均包含 10KB。

按照这个速度,我们每天将存储 10GB 的数据。

$$1M * 10KB \Rightarrow 10 \text{ GB/天}$$

如果我们想存储这些数据十年,我们需要 36TB 的总存储容量。

每天有 100 万个粘贴,10 年后我们将拥有 36 亿个粘贴。我们需要生成并存储密钥来唯一标识这些粘贴。如果我们使用 Base64 编码 ([AZ, az, 0-9, ., -]),我们将需要六个字母的字符串:

$$64^6 \approx 687 \text{ 亿个唯一字符串}$$

如果需要 1 个字节存储 1 个字符,则存储 3.6B 个密钥所需的总大小为:

$$3.6B * 6 \Rightarrow 22\text{GB}$$

与 36TB 相比,22GB 可以忽略不计。为了保留一定的余量,我们将假设 70% 的容量模型 (意味着我们在任何时候都不想使用超过 70% 的总存储容量),这会将我们的存储需求提高到 51.4TB。

带宽估计: 对于写入请求,我们预计每秒 12 个新粘贴,导致每秒 120KB 的入口。

$$12 * 10KB \Rightarrow 120KB/\text{秒}$$

至于读取请求,我们预计每秒有 58 个请求。因此,总数据输出 (发送给用户) 将为 0.6 MB/s。

$$58 * 10KB \Rightarrow 0.6 \text{ MB/秒}$$

尽管入口和出口总量并不大,但我们在设计服务时应该牢记这些数字。

内存估算:我们可以缓存一些经常访问的热贴。遵循 80-20 规则,即 20% 的热门粘贴产生 80% 的流量,我们希望缓存这 20% 的粘贴

由于我们每天有 500 万个读取请求,为了缓存这些请求的 20%,我们需要:

$$0.2 * 5M * 10KB \approx 10 \text{ GB}$$

5. 系统API

我们可以使用 SOAP 或 REST API 来公开我们服务的功能。以下是创建/检索/删除粘贴的 API 的定义:

`addPaste (api_dev_key,paste_data,custom_url =无用户名=无,paste_name =无,expire_date =无)`

参数:

`api_dev_key (string)`: 注册账户的API开发者密钥。除其他外,这将用于根据分配的配额限制用户。

`Paste_data` (字符串) :粘贴的文本数据。

`custom_url` (字符串) :可选的自定义 URL。

`user_name` (字符串) :用于生成 URL 的可选用户名。

`Paste_name` (字符串) :粘贴的可选名称

`expire_date` (字符串) :粘贴的可选到期日期。

返回: (字符串)

成功插入将返回可访问粘贴的 URL,否则将返回错误代码。

同样,我们可以检索和删除粘贴 API:

`getPaste(api_dev_key, api_paste_key)`

其中 “`api_paste_key`”是一个字符串,表示要检索的粘贴的粘贴密钥。该 API 将

返回粘贴的文本数据。

`deletePaste (api_dev_key,api_paste_key)`

成功删除返回 “`true`”,否则返回 “`false`”。

6. 数据库设计

关于我们存储的数据的性质的一些观察:

1. 我们需要存储数十亿条记录。
2. 我们存储的每个元数据对象都会很小 (小于 100 字节)。
3. 我们存储的每个粘贴对象可以是中等大小 (可以是几MB)。
4. 记录之间没有关系,除非我们想存储哪个用户创建了什么粘贴。

5.我们的服务是重读的。

数据库架构：

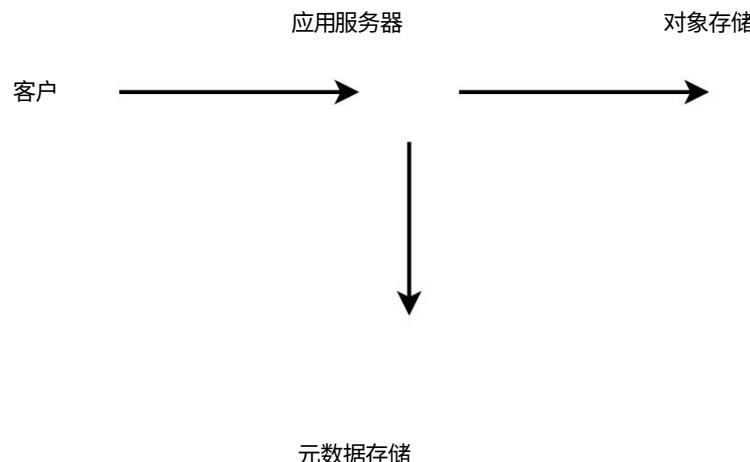
我们需要两个表,一个用于存储有关粘贴的信息,另一个用于存储用户数据。

粘贴		用户	
PK	URLHash:varchar(16)	PK	
内容密钥:varchar(512)		名称:varchar(20)	
到期日期:数据时间		电子邮件:varchar(32)	
[观众不支持]		创建日期:日期时间 	
创建日期:日期时间		上次登录:数据时间	

这里,“URLHash”是与 TinyURL 等效的 URL,“ContentKey”是存储粘贴内容的对象键。

7. 高层设计

在较高的层面上,我们需要一个应用程序层来服务所有的读写请求。应用层将与存储层通信以存储和检索数据。我们可以将存储层隔离,一个数据库存储与每个粘贴、用户等相关的元数据,而另一个数据库将粘贴内容存储在某些对象存储中(例如Amazon S3)。这种数据划分还允许我们单独缩放它们。



8. 组件设计

A. 应用层

我们的应用程序层将处理所有传入和传出的请求。应用程序服务器将与后端数据存储组件通信以服务请求。

如何处理写请求？收到写入请求后，我们的应用程序服务器将生成一个六字母的随机字符串，该字符串将用作粘贴的密钥（如果用户未提供自定义密钥）。然后应用程序服务器会将粘贴的内容和生成的密钥存储在数据库中。插入成功后，服务器可以将密钥返回给用户。这里一个可能的问题是插入由于重复的键而失败。由于我们生成随机密钥，因此新生成的密钥有可能与现有密钥匹配。在这种情况下，我们应该重新生成一个新密钥并重试。我们应该不断重试，直到看不到由于重复密钥而导致的失败。如果用户提供的自定义密钥已存在于我们的数据库中，我们应该向用户返回错误。

上述问题的另一个解决方案是运行一个独立的密钥生成服务（KGS），该服务预先生成随机的六个字母字符串并将它们存储在数据库中（我们称之为密钥数据库）。

每当我们想要存储新的粘贴时，我们只需获取已生成的密钥之一并使用它即可。

这种方法将使事情变得非常简单和快速，因为我们不会担心重复或冲突。KGS 将确保插入 key-DB 中的所有密钥都是唯一的。KGS 可以使用两张表来存储密钥，一张用于尚未使用的密钥，一张用于所有已使用的密钥。一旦 KGS 向应用程序服务器提供一些密钥，它就可以将这些密钥移动到已使用的密钥表中。KGS 可以始终将一些密钥保留在内存中，以便每当服务器需要它们时，它可以快速提供它们。一旦 KGS 在内存中加载一些密钥，它就可以将它们移动到已使用的密钥表中，这样我们就可以确保每个服务器都获得唯一的密钥。如果 KGS 在使用内存中加载的所有密钥之前就死掉了，我们将浪费这些密钥。鉴于我们有大量的密钥，我们可以忽略这些密钥。

KGS 不是单点故障吗？是的。为了解决这个问题，我们可以拥有一个 KGS 的备用副本，每当主服务器挂掉时，它就可以接管生成和提供密钥。

每个应用程序服务器可以缓存密钥数据库中的一些密钥吗？是的，这肯定可以加快速度。尽管在这种情况下，如果应用程序服务器在消耗所有密钥之前死亡，我们最终将丢失这些密钥。这是可以接受的，因为我们有 68B 个独特的六个字母键，这比我们需要的多得多。

它如何处理粘贴读取请求？收到读取粘贴请求后，应用程序服务

层联系数据存储。数据存储区搜索密钥，如果找到，则返回粘贴的内容。否则，返回错误代码。

b. 数据存储层

我们可以将数据存储层分为两层：

1. 元数据数据库:我们可以使用关系数据库,例如MySQL或分布式键值像 Dynamo 或 Cassandra 这样的存储。
2. 对象存储:我们可以将内容存储在对象存储中,例如 Amazon 的 S3。每当我们感觉想要达到内容存储的全部容量,我们可以通过添加更多内容来轻松增加它服务器。

Pastebin 的详细组件设计

9. 清除或数据库清理

请参阅设计 URL 缩短服务。

10. 数据分区和复制

请参阅设计 URL 缩短服务。

11. 缓存和负载均衡器

请参阅设计 URL 缩短服务。

12. 安全和权限

请参阅设计 URL 缩短服务。

设计Instagram

让我们设计一个像 Instagram 这样的照片共享服务,用户可以上传照片与其他用户共享。类似服务:Flickr、Picasa 难度级别:中

1. Instagram是什么?

Instagram 是一项社交网络服务,用户可以通过它上传照片和视频并与其它用户分享。Instagram 用户可以选择公开或私下分享信息。

任何其他用户都可以看到公开共享的任何内容,而私人共享的内容只能由指定的一组人访问。Instagram 还允许用户通过许多其他社交网络平台进行分享,例如 Facebook、Twitter、Flickr 和 Tumblr。

为了这个练习,我们计划设计一个更简单的 Instagram 版本,用户可以在其中分享照片,也可以关注其它用户。每个用户的“新闻源”将包含该用户关注的所有人的热门照片。

2. 系统的要求和目标

在设计 Instagram 时,我们将重点关注以下一组要求:

功能要求

1. 用户应该能够上传/下载/查看照片。
2. 用户可以根据照片/视频标题进行搜索。
3. 用户可以关注其它用户。
4. 系统应该能够生成并显示由热门照片组成的用户动态消息
来自用户关注的所有人。

非功能性需求

1. 我们的服务需要高可用。
2. 系统可接受的News Feed生成延迟为200ms。
3. 如果用户长时间看不到照片,一致性可能会受到影响(为了可用性)
尽管;应该没问题。
4. 系统应具有高可靠性;任何上传的照片或视频都不应丢失。

不在范围内:向照片添加标签、在标签上搜索照片、评论照片、将用户标记为照片、关注谁等。

3. 一些设计考虑

该系统的读取量很大,因此我们将专注于构建一个可以快速检索照片的系统。

1. 实际上,用户可以上传任意数量的照片。高效的存储管理
应该是设计该系统时的一个关键因素。

2. 查看照片时预计延迟较低。
3. 数据应100%可靠。如果用户上传照片,系统将保证它会永远不会迷失。

4. 容量估计和约束

- 假设我们有 5 亿总用户 ,其中每日活跃用户为 100 万。 · 每天 200 万张新照片 , 每秒 23 张新照片。 · 平均照片文件大小 => 200KB · 1 天照片所需的总空间

$$2M * 200KB \Rightarrow 400GB$$

- 10 年所需的总空间：

$$400GB * 365 \text{ (一年中的天数)} * 10 \text{ (年)} \approx 1425TB$$

5. 高层系统设计

在高层 ,我们需要支持两种场景 ,一种是上传照片 ,另一种是查看/搜索照片。我们的服务需要一些[对象存储](#)用于存储照片的服务器以及一些用于存储有关照片的元数据信息的[数据库服务器](#)。

6. 数据库架构

在面试的早期阶段定义数据库模式将有助于理解数据各个组件之间的流动 ,随后将指导数据分区。

我们需要存储有关用户、他们上传的照片以及他们关注的人的数据。照片表将存储与照片相关的所有数据 ;我们需要在 (PhotoID, CreationDate) 上有一个索引 ,因为我们需要获取首先是最近的照片。

存储上述模式的一个简单方法是使用像 MySQL 这样的 RDBMS,因为我们需要连接。但关系数据库也面临着挑战,尤其是当我们需要扩展它们时。有关详细信息,请查看[SQL 与 NoSQL](#)。

我们可以将照片存储在[HDFS](#)等分布式文件存储中或S3。

我们可以将上述模式存储在分布式键值存储中,以享受 NoSQL 提供的好处。

所有与照片相关的元数据都可以进入一个表,其中“键”是“PhotoID”,“值”是包含 PhotoLocation、UserLocation、CreationTimestamp 等的对象。

我们需要存储用户和照片之间的关系,以了解谁拥有哪张照片。我们还需要存储用户关注的人员列表。对于这两个表,我们可以使用像 Cassandra 这样的宽列数据存储。对于“UserPhoto”表,“键”将是“UserID”,“值”将是用户拥有的“PhotoID”列表,存储在不同的列中。我们将为“UserFollow”表提供类似的方案。

Cassandra 或键值存储通常始终维护一定数量的副本以提供可靠性。此外,在此类数据存储中,删除不会立即应用,数据会保留某些天(以支持取消删除),然后从系统中永久删除。

7. 数据大小估计

让我们估计一下每个表将有多少数据以及 10 年内需要多少总存储空间。

User:假设每个“int”和“dateTime”都是四个字节,则用户表中的每一行将有 68 个字节:

用户 ID (4 字节)+ 姓名 (20 字节)+ 电子邮件 (32 字节)+ 出生日期 (4 字节)+ 创建日期 (4 字节)+ 最后登录 (4 字节)= 68 字节

如果我们有 5 亿用户,我们将需要 32GB 的总存储空间。

$$5\text{亿} \times 68 \approx 32\text{GB}$$

Photo: Photo 表中的每一行均为 284 字节:

$$\begin{aligned} \text{PhotoID (4 字节)} + \text{UserID (4 字节)} + \text{PhotoPath (256 字节)} + \text{PhotoLatitude (4 字节)} + \\ \text{照片经度 (4 字节)} + \text{用户纬度 (4 字节)} + \text{用户经度 (4 字节)} + \text{创建日期 (4 字节)} = \\ 284 \text{字节} \end{aligned}$$

如果每天上传 200 万张新照片,我们一天需要 0.5GB 的存储空间:

$$2M * 284 \text{ 字节} \approx 0.5\text{GB 每天10 年内}$$

我们将需要 1.88TB 的存储空间。

UserFollow: UserFollow 表中的每一行由 8 个字节组成。如果我们有 5 亿用户,平均每个用户关注 500 个用户。我们需要 1.82TB 的存储空间用于 UserFollow 表:

$$\text{用户} * 500 \text{ 个关注者} * 8 \text{ 字节} \approx 1.82\text{TB 5 亿}$$

10 年所有表所需的总空间将为 3.7TB:

$$32\text{GB} + 1.88\text{TB} + 1.82\text{TB} \approx 3.7\text{TB}$$

8. 组件设计

照片上传 (或写入)可能会很慢,因为它们必须写入磁盘,而读取会更快,尤其是从缓存提供服务时。

上传用户可能会消耗所有可用连接,因为上传是一个缓慢的过程。这意味着如果系统忙于处理所有写入请求,则无法提供“读取”服务。我们应该牢记

在设计我们的系统之前,网络服务器有连接限制。如果我们假设一个 Web 服务器在任何时候最多可以有 500 个连接,那么它的并发上传或读取就不能超过 500 个。为了解决这个瓶颈,我们可以将读取和写入拆分为单独的服务。我们将拥有专门的读取服务器和不同的写入服务器,以确保上传不会占用系统。

分离照片的读取和写入请求还将使我们能够独立地扩展和优化每个操作。

9. 可靠性和冗余性

我们的服务不允许丢失文件。因此,我们将存储每个文件的多个副本,以便如果一台存储服务器挂掉了,我们可以从不同存储上的另一个副本中检索照片服务器。

同样的原理也适用于系统的其他组件。如果我们想要高可用性在系统中,我们需要在系统中运行多个服务副本,以便在少数服务停止运行时系统仍然可用并运行。冗余消除了系统中的单点故障。

如果某个服务在任何时候只需要运行一个实例,我们可以运行该服务的冗余辅助副本,该副本不提供任何流量,但当主服务器具有故障转移后,它可以在故障转移后接管控制权。
问题。

在系统中创建冗余可以消除单点故障,并在危机中需要时提供备份或备用功能。例如,如果同一服务有两个实例在运行
当生产系统出现故障或性能下降时,系统可以故障转移到健康副本。故障转移可以自动发生或需要手动干预。

10. 数据分片

让我们讨论元数据分片的不同方案：

A。基于 UserID 的分区让我们假设我们基于 “UserID”进行分片 ,以便我们可以将用户的所有照片保留在同一个分片上。如果一个数据库分片为 1TB ,我们将需要四个分片来存储 3.7TB 的数据。假设为了获得更好的性能和可扩展性 ,我们保留 10 个分片。

因此,我们将通过 UserID % 10 找到分片编号 ,然后将数据存储在那里。为了唯一地标识我们系统中的任何照片 ,我们可以为每个 Photoid 附加分片编号。

我们如何生成 Photoid?每个数据库分片都可以有自己的 Photoid 自动递增序列 ,并且由于我们将在每个 Photoid 后附加 ShardID ,这将使其在整个我们的数据库中都是唯一的。系统。

这种分区方案有哪些不同的问题?

1. 如何处理热点用户?有几个人关注这样的热门用户和很多其他人
查看他们上传的任何照片。
2. 有些用户会比其他人有很多照片 ,从而造成不统一
存储分配。
3. 如果我们无法将用户的所有图片存储在一个分片上怎么办?如果我们分发用户的照片
到多个分片上会导致更高的延迟吗?
4. 将用户的所有照片存储在一个分片上可能会导致一些问题 ,例如 ,如果该分片出现故障 ,则所有用户的数
据将不可用 ;如果该分片提供高负载服务 ,则延迟会更高等。

b. 基于 Photoid 的分区如果我们能够先生成唯一的 Photoid ,然后通过 “Photoid % 10”找到分片编号 ,那么上述
问题就迎刃而解了。在这种情况下 ,我们不需要将 ShardID 与 Photoid 一起附加 ,因为 Photoid 本身在整个系统中
是唯一的。

我们如何生成 PhotoID?这里我们不能在每个分片中使用自动递增序列来定义 PhotoID,因为我们需要首先知道 PhotoID 才能找到存储它的分片。—

解决方案可能是我们专用一个单独的数据库实例来生成自动递增的 ID。如果我们的 PhotoID 可以容纳 64 位,我们就可以定义一个仅包含 64 位 ID 字段的表。因此,每当我们想在系统中添加照片时,我们都可以在此表中插入一个新行,并将该 ID 作为新照片的 PhotoID。

这个生成数据库的密钥不会成为单点故障吗?是的,会的。一种解决方法是定义两个这样的数据库,其中一个生成偶数编号的 ID,另一个生成奇数编号的 ID。对于 MySQL,以下脚本可以定义这样的序列:

密钥生成服务器1:

自动增量增量 = 2
自动增量偏移 = 1

密钥生成服务器2:

自动增量增量 = 2
自动增量偏移 = 2

我们可以在这两个数据库前面放置一个负载均衡器,以在它们之间进行循环并处理停机时间。这两台服务器可能会不同步,其中一台生成的密钥多于另一台,但这不会在我们的系统中造成任何问题。我们可以通过定义单独的 ID 表来扩展此设计

对于用户、照片评论或我们系统中存在的其他对象。

或者,我们可以实现类似于我们在[设计类似 TinyURL 的 URL 缩短服务](#)中讨论的“密钥”生成方案。

我们如何规划系统的未来发展?我们可以拥有大量的逻辑分区来适应未来的数据增长,这样在开始时,多个逻辑分区驻留在单个物理数据库服务器上。由于每个数据库服务器上可以有多个数据库实例,因此我们可以为任何服务器上的每个逻辑分区拥有单独的数据库。所以每当我们感觉某个特定的数据库服务器有很多数据时,我们就可以从中迁移一些逻辑分区

到另一台服务器。我们可以维护一个配置文件(或一个单独的数据库),它可以将我们的逻辑分区映射到数据库服务器;这将使我们能够轻松移动分区。每当我们想要移动分区时,我们只需更新配置文件即可宣布更改。

11. 排名和动态消息生成

要为任何给定用户创建新闻源,我们需要获取用户关注的人的最新、最受欢迎和相关的照片。

为简单起见,我们假设我们需要获取用户新闻源的前 100 张照片。我们的应用程序服务器将首先获取用户关注的人员列表,然后从每个用户获取最新 100 张照片的元数据信息。在最后一步中,服务器会将所有这些照片提交给我们的排名算法,该算法将确定前 100 张照片(基于新近度、相似度等)并将其返回给用户。这种方法可能存在的问题是延迟较高,因为我们必须查询多个表并对结果执行排序/合并/排名。为了提高效率,我们可以预先生成 News Feed,并将其存储在单独的表中。

预生成新闻源:我们可以拥有专用服务器,不断生成用户的新闻源并将其存储在“UserNewsFeed”表中。因此,每当任何用户需要为其动态消息提供最新照片时,我们都会简单地查询此表并将结果返回给用户。

每当这些服务器需要生成用户的新闻源时,它们将首先查询 UserNewsFeed 表以查找上次为该用户生成新闻源的时间。然后,将从那时起生成新的新闻源数据(按照上述步骤)。

向用户发送动态消息内容有哪些不同的方法?

1. **拉取:**客户端可以定期或在需要时手动从服务器拉取 News Feed 内容。这种方法可能存在的问题是 a) 新数据可能不会显示给

b) 大多数情况下,如果没有新数据,拉取请求将导致空响应。

2. **推送:**服务器可以将新数据推送给用户。为了有效地管理这一点,用户必须维护[长轮询](#)向服务器请求接收更新。这种方法可能存在的问题是,关注很多人的用户或拥有数百万关注者的名人用户;在这种情况下,服务器必须非常频繁地推送更新。

3. **混合:**我们可以采用混合的方法。我们可以将所有拥有大量关注的用户转移到基于拉动的模型,并且只将数据推送给那些拥有数百(或数千)关注的用户。另一种方法可能是服务器向所有用户推送更新的频率不超过一定频率,让关注/更新较多的用户定期拉取数据。

有关动态消息生成的详细讨论,请查看[设计 Facebook 动态消息](#)。

12. 使用分片数据创建新闻源

为任何给定用户创建新闻源的最重要要求之一是获取该用户关注的所有人的最新照片。为此,我们需要有一种机制来按照片的创建时间对其进行排序。为了有效地做到这一点,我们可以将照片创建时间作为 PhotoID 的一部分。由于我们将在 PhotoID 上建立主索引,因此可以很快找到最新的 PhotoID。

我们可以为此使用纪元时间。假设我们的 PhotoID 将由两部分组成:第一部分将表示纪元时间,第二部分将是自动递增序列。因此,要创建新的 PhotoID,我们可以获取当前纪元时间并从密钥生成数据库中附加一个自动递增的 ID。我们可以从这个 PhotoID (PhotoID % 10) 中找出分片编号并存储照片

那里。

我们的 PhotoID 的大小是多少?假设我们的纪元时间从今天开始,我们需要多少位来存储未来 50 年的秒数?

$$86400 \text{ 秒/天} * 365 \text{ (一年天)} * 50 \text{ (年)} \Rightarrow 16 \text{ 亿秒}$$

我们需要 31 位来存储这个数字。因为平均而言 , 我们预计每秒 23 张新照片 ; 我们可以分配 9 位来存储自动递增序列。所以每一秒我们都可以存储 ($2^9 = 512$) 张新照片。我们可以每秒重置自动递增序列。

我们将在设计 Twitter 中的 “ 数据分片 ” 部分讨论有关此技术的更多细节。

13. 缓存和负载平衡

我们的服务需要一个大规模的照片传输系统来为全球分布的用户提供服务。

我们的服务应该使用大量地理上分布的照片缓存服务器并使用 CDN (有关详细信息 , 请参阅 [缓存](#)) 将其内容推送到更接近用户的位置。

我们可以为元数据服务器引入缓存来缓存热数据库行。我们可以使用 Memcache 来缓存数据 , 应用程序服务器在访问数据库之前可以快速检查缓存中是否有所需的行。最近最少使用 (LRU) 对于我们的系统来说是一个合理的缓存驱逐策略。

根据此策略 , 我们首先丢弃最近最少查看的行。

如何构建更加智能的缓存 ? 如果我们遵循 80-20 规则 , 即每日照片阅读量的 20% 会产生 80% 的流量 , 这意味着某些照片非常受欢迎 , 以至于大多数人都会阅读它们。这表明我们可以尝试缓存每日读取量的 20% 的照片和元数据。

设计 Dropbox

让我们设计一个文件托管服务 , 例如 Dropbox 或 Google Drive 。云文件存储使用户能够将数据存储在远程服务器上。通常 , 这些服务器由云存储提供商维护 , 并通过网络 (通常通过互联网) 提供给用户。用户按月支付云数据存储费用。类似服务 : OneDrive 、 Google Drive 难度级别 : 中

1. 为什么选择云存储 ?

云文件存储服务最近变得非常流行 , 因为它们简化了多个设备之间数字资源的存储和交换。从使用单一个人电脑到使用具有不同平台和操作系统的多个设备 (例如智能手机和平板电脑) 的转变 , 每个设备都可以随时从不同地理位置进行便携式访问 , 这被认为是云存储服务如此普及的原因。以下是一些主要好处

此类服务 :

可用性 : 云存储服务的座右铭是随时随地提供数据可用性。用户可以随时随地从任何设备访问他们的文件 / 照片。

可靠性和持久性 : 云存储的另一个好处是它提供 100% 的数据可靠性和持久性。云存储通过保留多份副本确保用户永远不会丢失数据

存储在不同地理位置的服务器上的数据。

可扩展性:用户永远不必担心存储空间不足。使用云存储,只要您准备好付费,您就可以拥有无限的存储空间。

如果您还没有使用过[dropbox.com](#)之前,我们强烈建议在那里创建一个帐户并上传/编辑文件,并浏览他们的服务提供的不同选项。这将对你理解本章有很大帮助。

2. 系统的要求和目标

您应该始终在面试开始时澄清要求。一定要问清楚

问题来找出面试官心目中系统的确切范围。

我们希望通过云存储系统实现什么目标?以下是我们系统的顶级要求:

1. 用户应该能够从任何设备上传和下载文件/照片。
2. 用户应该能够与其他用户共享文件或文件夹。
3. 我们的服务应该支持设备之间的自动同步,即更新文件后
在一台设备上,它应该在所有设备上同步。
4. 系统应支持存储最大1GB的大文件。
5. 需要酸性。应保证所有文件操作的原子性、一致性、隔离性和持久性。

6. 我们的系统应该支持离线编辑。用户应该能够添加/删除/修改文件
离线,一旦上线,所有更改都应同步到远程服务器和其他在线设备。

扩展要求

- 系统应支持数据快照,以便用户可以返回到任何版本的文件。

3. 一些设计考虑

- 我们应该预期巨大的读写量。 · 预计读写比率几乎相同。 ·
在内部,文件可以以小部分或块的形式存储(例如4MB);这可以提供
很多

好处是,所有失败的操作只能针对文件的较小部分重试。如果用户上传文件失败,则只会重试失败的块。

- 我们可以通过仅传输更新的块来减少数据交换量。 · 通过删除重复的块,我们可以节省存储空间和带宽
使用。 · 与客户端保存元数据(文件名、大小等)的本地副本可以为我们节省大量时间

- 到服务器的往返。 · 对于小
的更改,客户端可以智能地上传差异而不是整个块。

4. 容量估计和约束

- 假设我们的总用户数为 5 亿,每日活跃用户 (DAU) 为 1 亿。 · 假设平均每个用户通过三个不同的设备进行连接。
- 平均而言,如果用户拥有 200 个文件/照片,则我们将拥有 1000 亿个文件总数。
- 假设平均文件大小为100KB,这将为我们提供10 PB 的总存储空间。

$$100B * 100KB \Rightarrow 10PB$$

- 我们还假设每分钟有一百万个活动连接。

Machine Translated by Google

5. 高层设计

用户将指定一个文件夹作为其设备上的工作区。放置在此文件夹中的任何文件/照片/文件夹都将上传到云端，并且每当修改或删除文件时，都会以相同的方式反映在云存储中。用户可以在其所有设备和任何设备上指定类似的工作区

在一台设备上完成的修改将传播到所有其他设备，以便在任何地方都具有相同的工作空间视图。

在较高层面上，我们需要存储文件及其元数据信息，例如文件名、文件大小、目录等，以及与谁共享该文件。因此，我们需要一些服务器可以帮助客户端上传/下载文件到云存储，以及一些服务器可以帮助更新有关文件和用户的元数据。我们还需要某种机制来在更新发生时通知所有客户端，以便他们

可以同步他们的文件。

如下图所示，区块服务器将与客户端一起上传/下载文件

云存储和元数据服务器将在 SQL 或 NoSQL 数据库中更新文件的元数据。

同步服务器将处理通知所有客户端有关同步的不同更改的工作流程。

Dropbox 的高级设计

6. 组件设计

让我们一一浏览一下我们系统的主要组件：

A。客户

客户端应用程序监视用户计算机上的工作区文件夹，并将其中的所有文件/文件夹与远程云存储同步。客户端应用程序将与存储服务器一起上传，

下载并修改实际文件到后端云存储。客户端还与远程交互

同步服务处理任何文件元数据更新,例如文件名、大小、修改日期等的更改。

以下是客户的一些基本操作:

- 1.上传和下载文件。
- 2.检测工作区文件夹中的文件更改。
- 3.处理由于离线或并发更新而导致的冲突。

我们如何有效地处理文件传输?如上所述,我们可以将每个文件分成更小的块,以便我们只传输那些被修改的块,而不是整个文件。假设我们将每个文件划分为固定大小的 4MB 块。我们可以根据 1) 我们在云中使用的存储设备来静态计算最佳块大小,以优化空间利用率和每秒输入/输出操作 (IOPS) 2) 网络带宽 3) 存储中的平均文件大小等。

在我们的元数据中,我们还应该保留每个文件及其构成块的记录。

我们应该与客户保留一份元数据副本吗?保留元数据的本地副本不仅使我们能够进行离线更新,而且还节省了更新远程元数据的大量往返次数。

客户如何有效地倾听其他客户发生的变化?一种解决方案是客户端定期检查服务器是否有任何更改。这种方法的问题在于,我们在本地反映更改时会出现延迟,因为与服务器在发生更改时通知服务器相比,客户端将定期检查更改。如果客户端频繁检查服务器是否有更改,则不仅会浪费带宽(因为服务器大多数时候必须返回空响应),而且还会使服务器保持忙碌。拉取信息

这种方式是不可扩展的。

解决上述问题的方法可能是使用 HTTP 长轮询。通过长轮询,客户端向服务器请求信息,并期望服务器可能不会立即响应。

如果服务器在收到轮询时没有为客户端提供新数据,则服务器不会发送空响应,而是保持请求打开并等待响应信息可用。

一旦有新信息,服务器立即向客户端发送 HTTP/S 响应,完成打开的 HTTP/S 请求。收到服务器响应后,客户端可以立即发出另一个服务器请求以进行将来的更新。

基于以上考虑,我们可以将客户分为以下四个部分:

I. 内部元数据数据库将跟踪所有文件、块、它们的版本以及它们在文件系统中的位置。

II. Chunker会将文件分割成更小的块,称为块。它还将负责从文件块中重建文件。我们的分块算法将检测文件中已被用户修改的部分,并仅将这些部分传输到云存储;这将为我们节省带宽和同步时间。

三.观察者将监视本地工作区文件夹并通知索引器（如下所述）用户执行的任何操作，例如当用户创建、删除或更新文件或文件夹时。观察者还侦听同步服务广播的其他客户端上发生的任何更改。

四.索引器将处理从观察器接收到的事件，并使用有关已修改文件块的信息更新内部元数据数据库。一旦块成功提交/下载到云存储，索引器将与远程同步服务通信，以将更改广播到其他客户端并更新远程元数据数据库。

客户端应该如何处理缓慢的服务器？如果服务器繁忙/无响应，客户端应呈指数级后退。这意味着，如果服务器响应太慢，客户端应该延迟重试，并且这种延迟应该呈指数级增长。

移动客户端是否应该立即同步远程更改？与桌面或 Web 客户端不同，移动客户端通常按需同步以节省用户的带宽和空间。

b.元数据数据库

元数据数据库负责维护有关文件/块、用户和工作区的版本控制和元数据信息。元数据数据库可以是关系数据库（例如 MySQL），也可以是 NoSQL 数据库服务（例如 DynamoDB）。无论数据库的类型如何，同步服务都应该能够使用数据库提供一致的文件视图，特别是当多个用户同时使用同一文件时。由于 NoSQL 数据存储

不支持 ACID 属性以支持可扩展性和性能，我们需要以编程方式将对 ACID 属性的支持合并到我们的同步服务的逻辑中，以防万一。

选择这种数据库。但是,使用关系数据库可以简化同步服务的实现,因为它们本身支持 ACID 属性。

元数据数据库应存储有关以下对象的信息：

1. 块
2. 文件
3. 用户
4. 设备
5. 工作区（同步文件夹）

C。同步服务

同步服务是处理客户端进行的文件更新并将这些更改应用到其他订阅客户端的组件。它还将客户端的本地数据库与远程元数据数据库中存储的信息同步。同步服务是系统架构中最重要的部分,因为它在管理元数据和同步用户文件方面发挥着关键作用。桌面客户端与同步服务进行通信,以从云存储获取更新,或将文件和更新发送到云存储以及可能的其他用户。如果客户端离线一段时间,它会在新更新上线后立即轮询系统以获取新更新。当同步服务收到更新请求时,它会检查元数据数据库的一致性,然后继续更新。随后,向所有订阅的用户或设备发送通知以报告文件更新。

同步服务的设计方式应使其在客户端和云存储之间传输更少的数据,以实现更好的响应时间。为了实现这一设计目标,同步服务可以采用差分算法来减少需要同步的数据量。我们不是将整个文件从客户端传输到服务器,反之亦然

只能传输文件的两个版本之间的差异。因此,仅传输文件中已更改的部分。这也减少了带宽消耗和云数据存储

最终用户。如上所述,我们将把文件分成 4MB 的块,并且仅传输修改后的块。服务器和客户端可以计算散列(例如,SHA-256)来查看是否更新块的本地副本。在服务器上,如果我们已经有一个具有相似哈希值的块(甚至来自另一个用户),我们不需要创建另一个副本,我们可以使用相同的块。稍后将在重复数据删除中详细讨论这一点。

为了能够提供高效且可扩展的同步协议,我们可以考虑在客户端和同步服务之间使用通信中间件。消息中间件应提供可扩展的消息队列和更改通知,以支持使用拉或推策略的大量客户端。这样,多个同步服务实例就可以从全局请求队列接收请求,通信中间件将能够平衡其负载。



d.消息队列服务

我们架构的一个重要部分是消息传递中间件,它应该能够处理大量请求。支持客户端和同步服务之间基于异步消息的通信的可扩展消息队列服务最适合我们的应用程序的要求。消息队列服务支持系统的分布式组件之间的异步且松散耦合的基于消息的通信。消息队列服务应该能够在高度可用、可靠和可扩展的队列中有效地存储任意数量的消息。

消息队列服务将在我们的系统中实现两种类型的队列。请求队列是一个全局队列,所有客户端都会共享它。客户端更新元数据数据库的请求将首先发送到请求队列,同步服务将从那里接收它来更新元数据。

对应于各个订阅客户端的响应队列负责传递

更新消息给每个客户端。由于消息一旦被客户端接收到就会从队列中删除,因此我们需要为每个订阅的客户端创建单独的响应队列以共享更新消息。

e.云/块存储

云/块存储存储用户上传的文件块。客户端直接与存储交互以发送和接收对象。将元数据与存储分离使我们能够使用云中或内部的任何存储。

Dropbox 的详细组件设计

7. 文件处理工作流程

下面的序列显示了当客户端 A 更新与客户端 B 和 C 共享的文件时应用程序组件之间的交互,因此它们也应该收到更新。

如果其他客户端在更新时未联机,则消息队列服务会将更新通知保留在单独的响应队列中,直到它们稍后联机。

1. 客户端A将块上传到云存储。
2. 客户端 A 更新元数据并提交更改。
3. 客户端 A 获得确认,并向客户端 B 和 C 发送有关更改的通知。
4. 客户端 B 和 C 接收元数据更改并下载更新的块。

8. 重复数据删除

重复数据删除是一种用于消除数据重复副本以提高存储利用率的技术。它还可以应用于网络数据传输,以减少必须发送的字节数。对于每个新传入的块,我们可以计算它的哈希值,并将该哈希值与现有块的所有哈希值进行比较,以查看存储中是否已经存在相同的块。

我们可以在系统中通过两种方式实现重复数据删除:

A。后处理重复数据删除

通过后处理重复数据删除,新的块首先存储在存储设备上,然后一些进程分析数据以查找重复项。好处是客户无需等待

哈希计算或查找在存储数据之前完成,从而确保存储性能不会下降。这种方法的缺点是 1) 我们将不必要地存储重复数据,尽管时间很短,2) 传输重复数据会消耗带宽。

b.在线重复数据删除

或者,当客户端在其设备上输入数据时,可以实时完成重复数据删除哈希计算。如果我们的系统识别出它已经存储的块,则只会在元数据中添加对现有块的引用,而不是块的完整副本。这种方法将为我们提供最佳的网络和存储利用率。

9. 元数据分区

为了横向扩展元数据数据库,我们需要对其进行分区,以便它可以存储数百万用户的信息和数十亿个文件/块。我们需要想出一个分区方案来划分和存储我们的数据位于不同的数据库服务器中。

1. 垂直分区:我们可以对数据库进行分区,以便存储与一个相关的表

一台服务器上的特定功能。例如,我们可以将所有与用户相关的表存储在一个数据库中,并将所有文件/块相关的表存储在另一个数据库中。尽管这种方法实施起来很简单,但也存在一些问题:

1. 我们还会遇到规模问题吗?如果我们有数万亿个块要存储并且我们的数据库怎么办

无法支持存储如此海量的记录?我们如何进一步对这些表进行分区?

2. 连接两个独立数据库中的两个表可能会导致性能和一致性问题。

我们需要多久连接一次用户表和文件表?

2. 基于范围的分区:如果我们根据文件路径的第一个字母将文件/块存储在单独的分区中会怎样?在这种情况下,我们将以字母“A”开头的所有文件保存在一个分区中,并将以字母“B”开头的文件保存到另一个分区中,依此类推。这种方法称为基于范围的分区。我们甚至可以将某些不常出现的字母组合到一个数据库分区中。

我们应该静态地提出这个分区方案,以便我们始终可以以可预测的方式存储/查找文件。

这种方法的主要问题是它可能导致服务器不平衡。例如,如果我们决定将所有以字母“E”开头的文件放入数据库分区,后来我们发现以字母“E”开头的文件太多了,以至于我们无法容纳它们到一个数据库分区。

3. 基于散列的分区:在这个方案中,我们对我们存储的对象进行散列,并根据这个散列,我们计算出该对象应该进入的数据库分区。在我们的例子中,我们可以采取

我们要存储的文件对象的“FileID”的哈希值,以确定文件将存储的分区。

我们的哈希函数会将对象随机分布到不同的分区中,例如,我们的哈希函数总是可以将任何ID映射到[1…256]之间的数字,并且这个数字将是分区
我们将存储我们的对象。

这种方法仍然会导致分区过载,这可以通过使用一致性哈希来解决。

10. 缓存

我们的系统中可以有两种缓存。为了处理热文件/块,我们可以引入块存储的缓存。我们可以使用像Memcached这样的现成解决方案可以存储整个块及其各自的ID/哈希值,并且块服务器在命中块存储之前可以快速检查缓存是否有所需的块。根据客户的使用模式,我们可以确定我们需要多少个缓存服务器

需要。高端商用服务器可以有144GB内存;一台这样的服务器可以缓存36K块。

哪种缓存替换策略最适合我们的需求?当缓存已满时,我们想用更新/更热的块替换一个块,我们会如何选择?最近最少使用(LRU)对于我们的系统来说是一个合理的策略。根据这个策略,我们首先丢弃最近最少使用的块。

加载类似地,我们可以为元数据数据库提供缓存。

11. 负载均衡器 (LB)

我们可以在系统中的两个地方添加负载均衡层:1)在客户端和块服务器之间
2)客户端和元数据服务器之间。最初,可以采用简单的循环方法

在后端服务器之间平均分配传入请求。该LB实现简单,不会引入任何开销。这种方法的另一个好处是,如果服务器死机,LB会将其从轮换中删除,并停止向其发送任何流量。循环负载均衡的一个问题是,它不会考虑服务器负载。如果服务器过载或速度缓慢,负载均衡器不会停止向该服务器发送新请求。为了解决这个问题,可以放置更智能的LB解决方案,定期查询后端服务器的负载并据此调整流量。

12. 安全、权限和文件共享

用户在云中存储文件时最关心的问题之一是数据的隐私和安全,特别是在我们的系统中,用户可以与其他用户共享他们的文件,甚至将它们公开以与所有人共享。为了解决这个问题,我们将在元数据数据库中存储每个文件的权限,以反映哪些文件对任何用户可见或可修改。

设计 Facebook Messenger

让我们设计一个像 Facebook Messenger 这样的即时消息服务,用户可以在其中发送短信通过网络和移动界面相互交流。

1.什么是 Facebook Messenger?

Facebook Messenger 是一款为其用户提供基于文本的即时消息服务的软件应用程序。 Messenger 用户可以通过手机和 Facebook 网站与 Facebook 好友聊天。

2. 系统的要求和目标

我们的 Messenger 应满足以下要求：

功能要求：

1. Messenger应该支持用户之间的一对一对话。
2. Messenger 应跟踪用户的在线/离线状态。
3. Messenger应该支持聊天记录的持久存储。

非功能性要求：

1. 用户应该拥有最小延迟的实时聊天体验。
- 2.我们的系统应该高度一致;用户应该能够在所有内容上看到相同的聊天记录他们的设备。
3. Messenger的高可用性是可取的;为了以下目的,我们可以容忍较低的可用性一致性。

扩展要求：

- 群组聊天 :Messenger 应支持多人在群组中相互交谈。
- 推送通知 :Messenger 应该能够在用户有新消息时通知他们离线。

3. 容量估计和约束

假设我们有 5 亿日活跃用户,平均每个用户每天发送 40 条消息;这每天给我们带来 200 亿条消息。

存储估算:假设一条消息平均为 100 字节,因此要存储一天的所有消息,我们需要 2TB 的存储空间。

$$200 \text{ 亿条消息} * 100 \text{ 字节} \Rightarrow 2 \text{ TB/天}$$

要存储五年的聊天历史记录,我们需要 3.6 PB 的存储空间。

$$2 \text{ TB} * 365 \text{ 天} * 5 \text{ 年} \approx 3.6 \text{ PB}$$

除了聊天消息之外,我们还需要存储用户信息、消息元数据 (ID、时间戳等)。更不用说,上述计算没有考虑数据压缩和复制。

带宽估计:如果我们的服务每天获取 2TB 的数据,那么每秒将为我们提供 25MB 的传入数据。

$$2 \text{ TB} / 86400 \text{ 秒} \approx 25 \text{ MB}/\text{秒}$$

由于每条传入消息都需要发送给另一个用户,因此我们需要相同的带宽 25MB/s 来进行上传和下载。

高水平估计:

每天消息总数 200 亿条

每天存储 2TB

存储 5 年 3.6PB

输入数据 25MB/秒

传出数据 25MB/秒

4. 高层设计

在高层,我们需要一个聊天服务器作为核心部分,协调用户之间的所有通信。当一个用户想要向另一个用户发送消息时,他们会连接

到聊天服务器并向服务器发送消息;然后,服务器将该消息传递给其他用户并将其存储在数据库中。

详细的工作流程如下所示:

1. 用户 A 通过聊天服务器向用户 B 发送消息。
2. 服务器接收消息并向用户 A 发送确认。
3. 服务器将消息存储在其数据库中并将消息发送给用户 B。
4. 用户 B 接收消息并向服务器发送确认。

5. 服务器通知用户A消息已成功传递给用户B。

发送消息的请求流程

1共 8 个

5. 详细组件设计

让我们首先尝试构建一个简单的解决方案,其中所有内容都在一台服务器上运行。在高层,我们的系统需要处理以下用例:

1. 接收传入消息并传递传出消息。
2. 在数据库中存储和检索消息。
3. 记录哪些用户在线或离线,并将这些状态变化通知所有相关用户。

我们来一一谈谈这些场景:

A. 消息处理

我们如何有效地发送/接收消息?要发送消息,用户需要连接到服务器并为其他用户发布消息。要从服务器获取消息,用户有两种选择:

1. **拉取模型**: 用户可以定期向服务器询问是否有新消息。
2. **推送模型**: 用户可以与服务器保持开放的连接,并且可以依赖服务器
每当有新消息时通知他们。

如果我们采用第一种方法,那么服务器需要跟踪仍在等待的消息

一旦接收用户连接到服务器请求任何新消息,服务器就可以返回所有待处理的消息。为了最大限度地减少用户的延迟,他们必须经常检查服务器,并且大多数时候,如果没有待处理的消息,他们将得到空响应。这会浪费大量资源,而且看起来并不是一个有效的解决方案。

如果我们采用第二种方法,即所有活动用户与服务器保持打开的连接,那么一旦服务器收到消息,它就可以立即将消息传递给目标用户。

这样,服务器不需要跟踪待处理的消息,并且我们将具有最小的延迟,因为消息是在打开的连接上立即传递的。

客户端如何与服务器保持开放连接?我们可以使用 [HTTP长轮询](#)或[WebSocket](#)。在长轮询中,客户端可以向服务器请求信息,但期望服务器可能不会立即响应。如果服务器在收到轮询时没有为客户端提供新数据,则服务器不会发送空响应,而是保持请求打开并等待

响应信息变得可用。一旦有新信息，服务器立即将响应发送给客户端，完成打开请求。收到服务器响应后，客户端可以立即发出另一个服务器请求以进行将来的更新。这带来了很多改进

延迟、吞吐量和性能。长轮询请求可能会超时或可能会收到与服务器的断开连接，在这种情况下，客户端必须打开新的请求。

服务器如何跟踪所有打开的连接以有效地将消息重定向到用户？服务器可以维护一个哈希表，其中“key”是用户 ID，“value”是

是连接对象。因此，每当服务器收到用户的消息时，它都会在哈希表中查找该用户以找到连接对象，并根据打开的请求发送消息。

当服务器收到用户下线的消息时会发生什么？如果接收方已断开连接，服务器可以通知发送方传送失败。如果是临时断开连接，例如接收方的长轮询请求刚刚超时，那么我们应该期望用户重新连接。在这种情况下，我们可以要求发件人重试发送消息。

此重试可以嵌入到客户端的逻辑中，以便用户不必重新键入消息。服务器还可以将消息存储一段时间，并在接收者重新连接后重试发送。

我们需要多少个聊天服务器？随时规划5亿连接。假设现代服务器可以随时处理 50K 并发连接，我们将需要 10K 这样的服务器。

我们如何知道哪个服务器拥有与哪个用户的连接？我们可以介绍一个软件

我们的聊天服务器前面的负载平衡器；它可以将每个 UserID 映射到服务器以重定向请求。

服务器应如何处理“传递消息”请求？服务器在收到新消息后需要执行以下操作：1) 将消息存储在数据库中 2) 将消息发送给接收者 3) 向发送者发送确认。

聊天服务器将首先找到为接收者保留连接的服务器，并将消息传递给该服务器以将其发送给接收者。然后聊天服务器可以将确认发送给发送者；我们不需要等待将消息存储在数据库中（这可以在后台发生）。下一节将讨论存储消息。

消息传递者如何保持消息的顺序？我们可以为每条消息存储一个时间戳，这是服务器接收消息的时间。这仍然不能确保客户端消息的正确排序。服务器时间戳无法确定消息的确切顺序的情况如下所示：

1. User-1 向 User-2 的服务器发送消息 M1。
2. 服务器在 T1 接收 M1。
3. 同时，User-2 向 User-1 的服务器发送消息 M2。
4. 服务器在 T2 接收消息 M2，使得 T2 > T1。
5. 服务器将消息 M1 发送给 User-2，将 M2 发送给 User-1。

因此，用户 1 将首先看到 M1，然后是 M2，而用户 2 将首先看到 M2，然后是 M1。

为了解决这个问题,我们需要为每个客户端的每条消息保留一个序列号。该序列号将确定每个用户的消息的确切顺序。通过此解决方案,两个客户端都将看到消息序列的不同视图,但此视图在所有设备上都将保持一致。

b.从数据库中存储和检索消息

每当聊天服务器收到新消息时,都需要将其存储在数据库中。为此,我们有两个选项:

1. 启动一个单独的线程,该线程将与数据库一起存储消息。
2. 向数据库发送异步请求来存储消息。

在设计数据库时,我们必须记住以下几点:

1. 如何高效地使用数据库连接池。
2. 如何重试失败的请求。
3. 在哪里记录那些重试后仍失败的请求。
4. 当所有问题都解决后,如何重试这些记录的请求(重试后失败的请求)。

我们应该使用哪种存储系统?我们需要有一个能够支持非常高速率的数据库

小更新并快速获取一系列记录。这是必需的,因为我们有大量的小消息需要插入到数据库中,并且在查询时,用户最感兴趣的是顺序访问这些消息。

我们无法使用 MySQL 等 RDBMS 或 MongoDB 等 NoSQL,因为我们无法在用户每次接收/发送消息时从数据库中读取/写入一行。这不仅会使我们服务的基本操作以高延迟运行,还会对数据库造成巨大的负载。

像 HBase 这样的宽列数据库解决方案就可以轻松满足我们的这两个要求。HBase 是一种面向列的键值 NoSQL 数据库,可以将一个键的多个值存储到多个列中。HBase 是仿照 Google BigTable 设计的并在 Hadoop 分布式文件系统 (HDFS) 之上运行。HBase 将数据分组在一起,将新数据存储在内存缓冲区中,一旦缓冲区已满,它将数据转储到磁盘。这种存储方式不仅有助于存储大量小数据

快速,而且还可以通过键或扫描行范围来获取行。HBase 也是一个高效的数据库,可以存储不同大小的数据,这也是我们的服务所需要的。

客户端应该如何高效地从服务器获取数据?客户端在从服务器获取数据时应该分页。对于不同的客户端,页面大小可能不同,例如,手机的屏幕较小,因此我们在视口中需要较少数量的消息/对话。

C.管理用户状态

我们需要跟踪用户的在线/离线状态,并在状态发生变化时通知所有相关用户。由于我们在服务器上为所有活动用户维护一个连接对象,因此我们可以

由此轻松了解用户当前的状态。任何时候都有 5 亿活跃用户,如果我们必须的话

将每个状态变化广播给所有相关的活跃用户 ,会消耗大量资源。围绕这一点我们可以做如下优化：

- 1.每当客户端启动应用程序时,它可以拉取好友列表中所有用户的当前状态。
- 2.每当一个用户向另一个离线的用户发送消息时,我们可以向发送者发送失败消息并更新客户端上的状态。
- 3.每当用户上线时,服务器总是可以延迟几秒广播该状态
秒查看用户是否没有立即离线。
- 4.客户端可以从服务器获取有关用户视口上显示的用户的状态。这不应该是一个频繁的操作,因为服务器正在广播用户的在线状态,我们可以忍受用户陈旧的离线状态一段时间。
- 5.每当客户端与另一个用户开始新的聊天时,我们就可以拉取当时的状态。

Facebook Messenger 的详细组件设计

设计总结:客户端会打开一个到聊天服务器的连接来发送消息;然后服务器会将其传递给请求的用户。所有活动用户都将保持与服务器的连接以接收消息。每当有新消息到达时,聊天服务器都会将其推送给接收用户

长轮询请求。消息可以存储在HBase中,支持快速小更新和范围

基于搜索。服务器可以向其他相关用户广播用户的在线状态。客户可以不那么频繁地为客户端视口中可见的用户拉取状态更新。

6. 数据分区

由于我们将存储大量数据（五年 3.6PB），因此我们需要将其分发到多个数据库服务器上。我们的分区方案是什么？

基于 UserID 的分区：假设我们基于 UserID 的哈希进行分区，以便我们可以将用户的所有消息保存在同一个数据库中。如果一个数据库分片是 4TB，那么五年内我们将拥有 “ $3.6\text{PB}/4\text{TB} \approx 900$ ” 个分片。为了简单起见，我们假设我们保留 1K 分片。因此，我们将通过 “ $\text{hash}(\text{UserID}) \% 1000$ ” 找到分片编号，然后从那里存储/检索数据。这种分区方案也可以非常快速地获取任何用户的聊天历史记录。

一开始，我们可以从较少的数据库服务器开始，在一台物理服务器上驻留多个分片。由于我们可以在一台服务器上拥有多个数据库实例，因此我们可以轻松地存储多个单个服务器上的分区。我们的哈希函数需要理解这种逻辑分区方案，以便它可以在一台物理服务器上映射多个逻辑分区。

由于我们将存储无限的消息历史记录，因此我们可以从大量逻辑分区开始，这些逻辑分区将映射到更少的物理服务器，并且随着存储需求的增加，我们可以添加更多物理服务器来分布我们的逻辑分区。

基于 MessageID 的分区：如果我们将一个用户的不同消息存储在不同的数据库分片上，那么获取一段聊天的一系列消息会非常慢，所以我们不应该采用这种方案。

7. 缓存

我们可以在用户视口中可见的一些最近对话（例如最后 5 条）中缓存一些最近的消息（例如最后 15 条）。由于我们决定将所有用户的消息存储在一个分片上，因此缓存用户也应该完全驻留在一台机器上。

8. 负载均衡

我们需要在聊天服务器前面有一个负载均衡器；它可以将每个 UserID 映射到保存用户连接的服务器，然后将请求定向到该服务器。同样，我们的缓存服务器需要一个负载平衡器。

9. 容错和复制

当聊天服务器出现故障时会发生什么？我们的聊天服务器与用户保持连接。如果服务器出现故障，我们是否应该设计一种机制将这些连接转移到其他服务器？将 TCP 连接故障转移到其他服务器非常困难；一种更简单的方法是让客户端在连接丢失时自动重新连接。

我们应该存储用户消息的多个副本吗？我们不能只有一份用户数据的副本，因为如果保存数据的服务器崩溃或永久关闭，我们没有任何机制来保存数据。

恢复该数据。为此,我们要么必须在不同的服务器上存储数据的多个副本,要么使用诸如里德-所罗门编码之类的技术来分发和复制它。

10. 扩展要求

A.群聊

我们可以在系统中拥有单独的群聊对象,这些对象可以存储在聊天服务器上。群聊对象由 GroupChatID 标识,并且还将维护属于该聊天的人员列表。我们的负载均衡器可以根据 GroupChatID 引导每个群聊消息,并且处理该群聊的服务器可以迭代所有聊天用户,找到处理每个用户连接的服务器来传递消息。

在数据库中,我们可以将所有群聊存储在一个根据GroupChatID分区的单独表中。

b.推送通知

在我们当前的设计中,用户只能向活动用户发送消息,如果接收用户离线,我们会向发送用户发送失败消息。推送通知将使我们的系统能够向离线用户发送消息。

对于推送通知,每个用户都可以选择从他们的设备(或网络浏览器)接收通知,只要有新消息或事件。每个制造商都维护一组服务器来处理将这些通知推送给用户。

要在我们的系统中拥有推送通知,我们需要设置一个通知服务器,这将需要离线用户的消息并将其发送到制造商的推送通知服务器,然后该服务器将其发送到用户的设备。

设计推特

让我们设计一个类似 Twitter 的社交网络服务。该服务的用户将能够发布推文、关注其他人以及喜欢的推文。难度级别:中等

1.什么是推特?

Twitter 是一种在线社交网络服务,用户可以在其中发布和阅读称为“推文”的 140 个字符的简短消息。注册用户可以发布和阅读推文,但未注册的用户只能阅读推文。用户通过网站界面、短信或移动应用程序访问 Twitter。

2. 系统的要求和目标

我们将设计一个更简单的 Twitter 版本,并满足以下要求:

功能要求

1. 用户应该能够发布新推文。
2. 用户应该能够关注其他用户。
3. 用户应该能够将推文标记为收藏夹。
4. 该服务应该能够创建并显示用户的时间线,其中包含用户关注的所有人员的热门推文。

5. 推文可以包含照片和视频。

非功能性需求

1. 我们的服务需要高可用。
2. 系统可接受的时间线生成延迟为200ms。
3. 一致性可能会受到影响（为了可用性）;如果用户没有看到某条推文
同时,应该没问题。

扩展要求

1. 搜索推文。
2. 回复推文。
3. 热门话题 – 当前热门话题/搜索。
4. 标记其他用户。
5. 推文通知。
6. 跟随谁?建议?
7. 时刻。

3. 容量估计和约束

假设我们的总用户数为 10 亿,其中每日活跃用户 (DAU) 为 2 亿。还假设我们每天有 1 亿条新推文,平均每个用户关注 200 人。

每天有多少个收藏夹?如果平均每个用户每天喜欢 5 条推文,我们将得到:

$$2\text{亿用户} * 5 \text{个收藏夹} \Rightarrow 1\text{B 个收藏夹}$$

我们的系统总共会产生多少推文浏览量?我们假设平均而言,用户每天访问其时间线两次,并访问其他五个人的页面。如果用户在每个页面上看到 20 条推文,那么我们的系统将生成 28B/天的总推文浏览量:

$$200\text{M DAU} * ((2 + 5) * 20 \text{条推文}) \Rightarrow 28\text{B/天}$$

存储估算假设每条推文有 140 个字符,我们需要两个字节来存储一个字符而不进行压缩。假设我们需要 30 个字节来存储每条推文的元数据(如 ID、时间戳、用户 ID 等)。我们需要的总存储空间:

$$100\text{M} * (280 + 30) \text{字节} \Rightarrow 30\text{GB/天}$$

五年内我们的存储需求是多少?我们需要多少存储空间来存储用户数据、关注数据、收藏夹?我们将把它留给练习。

并非所有推文都会有媒体,我们假设平均每五条推文有一张照片,每十分之一有一个视频。我们还假设平均一张照片为 200KB,一段视频为 2MB。这将引导我们每天拥有 24TB 的新媒体。

$$(100M/5 \text{ 张照片} * 200\text{KB}) + (100M/10 \text{ 个视频} * 2\text{MB}) \approx 24\text{TB/天}$$

带宽估计由于总入口量为每天 24TB,这将转化为 290MB/秒。

请记住,我们每天的推文浏览量为 28B。我们必须显示每条推文的照片 (如果它有照片),但我们假设用户观看他们在时间线中看到的每第三个视频。因此,总出口将为:

$$\begin{aligned} & (28B * 280 \text{ 字节}) / 86400 \text{ 条文本} \Rightarrow 93\text{MB/s} \\ & + (28B/5 * 200\text{KB}) / 86400 \text{ 张照片} \Rightarrow 13\text{GB/S} \\ & + (28B/10/3 * 2\text{MB}) / 86400 \text{ 视频} \Rightarrow 22\text{GB/s} \end{aligned}$$

$$\text{总计 } \approx 35\text{GB/s}$$

4. 系统API

一旦我们最终确定了需求,定义系统 API 总是一个好主意。这应明确说明系统的期望。

我们可以使用 SOAP 或 REST API 来公开我们服务的功能。以下是用于发布新推文的 API 的定义:

推文 (api_dev_key,tweet_data,tweet_location,user_location,media_ids,maximum_results_to_return)

参数:

api_dev_key (string): 注册账户的API开发者密钥。除其他外,这将用于根据分配的配额限制用户。

tweet_data (字符串) :推文的文本,通常最多 140 个字符。

tweet_location (字符串) :此推文所指的可选位置 (经度、纬度)。**user_location (字符串) :**添加推文的用户的可选位置 (经度、纬度)。

media_ids (number[]):与推文关联的可选 media_ids 列表。(所有媒体照片,视频等需要单独上传)。

返回: (字符串)

成功的帖子将返回访问该推文的 URL。否则,将返回适当的 HTTP 错误。

5. 高层系统设计

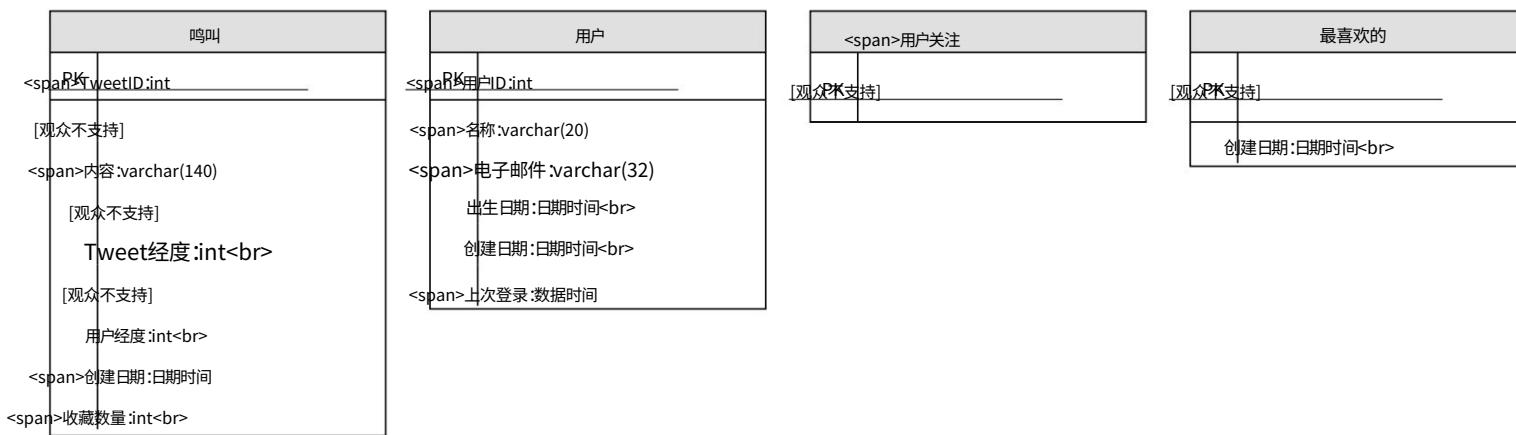
我们需要一个能够高效存储所有新推文的系统,100M/86400s => 每秒1150条推文
每秒读取 28B/86400s => 325K 条推文。从需求中可以清楚地看出,这将是一个读取繁重的系统。

在较高层面上,我们需要多个应用程序服务器来服务所有这些请求,并在它们前面提供负载均衡器以进行流量分配。在后端,我们需要一个高效的数据库,能够存储所有新的推文,并且能够支持海量的读取。我们还需要一些文件存储来存储照片和视频。

尽管我们预计每日写入负载为 1 亿条推文,读取负载为 280 亿条推文。这意味着我们的系统平均每秒将收到大约 1160 条新推文和 325K 读取请求。不过,该流量全天分布不均匀,但在高峰时段,我们预计每秒至少有几千个写入请求和大约 100 万个读取请求。我们在设计系统架构时应该牢记这一点。

6. 数据库架构

我们需要存储有关用户、他们的推文、他们最喜欢的推文以及他们关注的人的数据。



要在 SQL 和 NoSQL 数据库之间进行选择来存储上述架构,请参阅设计 Instagram 下的“数据库架构”。

7. 数据分片

由于我们每天都会有大量的新推文,并且读取负载也非常高,因此我们需要将数据分发到多台机器上,以便能够高效地读/写。我们有很多选择来分片我们的数据;让我们一道来:

基于 UserID 的分片:我们可以尝试将一个用户的所有数据存储在一台服务器上。在存储时,我们可以将 UserID 传递给哈希函数,该函数将用户映射到我们将存储的数据库服务器

用户的所有推文、收藏夹、关注等。在查询用户的推文/关注/收藏夹时,我们可以询问我们的哈希函数在哪里可以找到用户的数据,然后从那里读取它。这种方法有几个问题:

1. 如果用户变得热门怎么办?持有用户的服务器上可能有很多查询。这高负载会影响我们服务的性能。
2. 随着时间的推移,与其他用户相比,一些用户最终可能会存储大量推文或拥有大量关注者。保持不断增长的用户数据的均匀分布是相当困难的。

为了从这些情况中恢复,我们必须重新分区/重新分布我们的数据或使用一致的散列。

基于 TweetID 的分片:我们的哈希函数会将每个 TweetID 映射到一个随机服务器,我们将在其中存储该推文。要搜索推文,我们必须查询所有服务器,每个服务器都会返回一组推文。中央服务器将汇总这些结果并将其返回给用户。让我们看一下时间线生成示例;以下是我们的系统生成用户时间线必须执行的步骤数:

1. 我们的应用程序 (app) 服务器将找到用户关注的所有人员。
2. 应用服务器将查询发送到所有数据库服务器以查找这些人的推文。
3. 每个数据库服务器将查找每个用户的推文,按新近度排序并返回顶部推文。
4. 应用服务器将所有结果合并并重新排序,将排名靠前的结果返回给用户。

这种方式解决了热点用户的问题,但是与按 UserID 分片相比,我们必须查询所有数据库分区来查找用户的推文,这可能会导致更高的延迟。

我们可以通过在数据库服务器前面引入缓存来存储热门推文来进一步提高性能。

基于推文创建时间的分片:基于创建时间存储推文将为我们提供快速获取所有热门推文的优势,并且我们只需要查询非常小的一组服务器。

这里的问题是流量负载不会被分配,例如,在写入时,所有新推文将发送到一台服务器,而其余服务器将闲置。同样,在读取时,服务器

与保存旧数据的服务器相比,保存最新数据的负载将非常高。

如果我们可以将 TweetID 和推文创建时间的分片结合起来会怎么样?如果我们不单独存储推文创建时间并使用 TweetID 来反映这一点,我们就可以从这两种方法中获益。

这样就可以很快找到最新的推文。为此,我们必须使每个 TweetID 在我们的系统中普遍唯一,并且每个 TweetID 也应该包含一个时间戳。

我们可以为此使用纪元时间。假设我们的 TweetID 将有两部分:第一部分将表示纪元秒,第二部分将是自动递增序列。因此,要创建一个新的 TweetID,我们可以获取当前纪元时间并向其附加一个自动递增的数字。我们可以从这个 TweetID 中找出分片编号并将其存储在那里。

我们的 TweetID 的大小是多少?假设我们的纪元时间从今天开始,我们需要多少位需要存储未来50年的秒数?

$$86400 \text{ 秒/天} * 365 \text{ (一年天)} * 50 \text{ (年)} \Rightarrow 1.6B$$

我们需要 31 位来存储这个数字。由于我们平均每秒预计有 1150 条新推文,因此我们可以分配 17 位来存储自动递增序列;这将使我们的 TweetID 长为 48 位。因此,每秒我们可以存储 ($2^{17} \Rightarrow 130K$) 条新推文。我们可以每秒重置自动递增序列。为了容错和更好的性能,我们可以有两台数据库服务器为我们生成自增键,一台生成偶数键,另一台生成

奇数键。

如果我们假设当前的纪元秒是“1483228800”,我们的 TweetID 将如下所示:

1483228800 000001

1483228800 000002

1483228800 000003

1483228800 000004

.....

如果我们将 TweetID 设为 64 位(8 字节)长,我们就可以轻松存储未来 100 年的推文,并以毫秒为粒度进行存储。

在上述方法中,我们仍然需要查询所有服务器以生成时间线,但我们的读取(和写入)将会快得多。

1. 由于我们没有任何二级索引(在创建时),这将减少我们的写入延迟。
2. 在读取时,我们不需要过滤创建时间,因为我们的主键具有纪元时间包含在其中。

8. 缓存

我们可以为数据库服务器引入缓存来缓存热门推文和用户。我们可以使用一个现成的像 Memcache 这样的架子解决方案可以存储整个推文对象。应用服务器在访问数据库之前可以快速检查缓存中是否有所需的推文。根据客户的使用模式,我们可以确定需要多少个缓存服务器。

哪种缓存替换策略最适合我们的需求?当缓存已满并且我们想用更新/更热门的推文替换一条推文时,我们会如何选择?最近最少使用 (LRU)对于我们的系统来说是一个合理的策略。根据此政策,我们首先丢弃最近最少查看的推文。

如何才能拥有更智能的缓存呢?如果我们采用 80-20 规则,即 20% 的推文产生 80% 的阅读流量,这意味着某些推文非常受欢迎,以至于大多数人都会阅读它们。这表明我们可以尝试缓存每个分片每日读取量的 20%。

如果我们缓存最新的数据怎么办?我们的服务可以从这种方法中受益。假设我们 80% 的用户只看到过去三天的推文;我们可以尝试缓存过去三天的所有推文。假设我们有专用的缓存服务器,可以缓存过去三天所有用户的所有推文。根据上述估计,我们每天会收到 1 亿条新推文或 30GB 新数据 (不包括照片和视频)。如果我们想要存储过去三天的所有推文,我们将需要不到 100GB 的内存。这些数据可以很容易地放入一台服务器中,但我们应该将其复制到多台服务器上以分配所有读取流量,从而减少缓存服务器上的负载。因此,每当我们生成用户的时间线时,我们都可以询问缓存服务器是否拥有该用户的所有最新推文。如果是,我们可以简单地从缓存中返回所有数据。如果缓存中没有足够的推文,我们必须查询后端服务器来获取该数据。在类似的设计中,我们可以尝试缓存最近三天的照片和视频。

我们的缓存就像一个哈希表,其中 “key”是 “OwnerId” , “value”是一个双向链表,其中包含该用户在过去三天内的所有推文。由于我们希望首先检索最新的数据,因此我们始终可以在链表的头部插入新的推文,这意味着所有较旧的推文将位于链表的尾部附近。因此,我们可以从尾部删除推文,为新的推文腾出空间。

9. 时间线生成

有关时间线生成的详细讨论,请查看[设计 Facebook 的新闻源](#)。

10. 复制和容错

由于我们的系统是读取密集型的,因此我们可以为每个数据库分区拥有多个辅助数据库服务器。

辅助服务器将仅用于读取流量。所有写入将首先发送到主服务器,然后复制到辅助服务器。该方案还为我们提供了容错能力,因为每当主服务器出现故障时,我们都可以故障转移到辅助服务器。

11. 负载均衡

我们可以在系统中的三个位置添加负载平衡层:1)客户端和应用程序服务器之间,2)应用程序服务器和数据库复制服务器之间,3)聚合服务器和缓存服务器之间。最初,可以采用简单的循环方法;在服务器之间平均分配传入请求。该LB实现简单,不会引入任何开销。这种方法的另一个好处是,如果服务器死机,LB 会将其从轮换中删除,并停止向其发送任何流量。循环负载均衡的一个问题是它不会占用服务器

负载考虑。如果服务器过载或速度缓慢，负载均衡器不会停止向该服务器发送新请求。为了解决这个问题，可以放置更智能的 LB 解决方案，定期查询后端服务器的负载并据此调整流量。

12. 监控

拥有监控我们系统的能力至关重要。我们应该不断收集数据，以便立即了解我们的系统的运行情况。我们可以收集以下指标/计数器来了解

我们的服务表现：

1. 每天/秒的新推文数量，每日峰值是多少？
2. 时间线传送统计数据，我们的服务每天/每秒传送多少条推文。
3. 用户看到的刷新时间线的平均延迟。

通过监视这些计数器，我们将意识到是否需要更多复制、负载平衡或缓存。

13. 扩展要求

我们如何提供 Feed？获取某人关注的所有最新推文，并按时间对它们进行合并/排序。使用分页来获取/显示推文。只获取所有关注者的前 N 条推文。这个 N 将取决于客户端的视口，因为在移动设备上我们显示的推文较少

与 Web 客户端相比。我们还可以缓存下一个热门推文以加快速度。

或者，我们可以预生成 feed 以提高效率；有关详细信息，请参阅“设计 Instagram”下的“排名和时间线生成”。

转推：对于数据库中的每个推文对象，我们可以存储原始推文的 ID，并且在此转推对象上不存储任何内容。

热门主题：我们可以缓存最近 N 秒内最常出现的主题标签或搜索查询，并在每 M 秒后不断更新它们。我们可以根据推文、搜索查询、转发或点赞的频率对热门主题进行排名。我们可以给予以下主题更多的权重

展现给更多的人。

跟随谁？如何提出建议？此功能将提高用户参与度。我们可以建议

有人关注的朋友。我们可以下两三层去找名人提建议。我们可以优先考虑拥有更多关注者的人。

由于任何时候只能提出一些建议，因此请使用机器学习（ML）来调整并重新确定优先级。机器学习信号可能包括最近关注量增加的人、共同关注者（如果其他人正在关注该用户）、共同位置或兴趣等。

时刻：获取过去 1 或 2 小时内不同网站的热门新闻，找出相关推文，确定优先级

使用 ML（监督学习或聚类）对它们进行分类（新闻、支持、金融、娱乐等）。然后我们就可以把这些文章作为朋友圈的热门话题展示出来。

搜索：搜索涉及推文的索引、排名和检索。我们的下一个问题“设计 Twitter 搜索”中讨论了类似的解决方案。

设计 Youtube 或 Netflix

让我们设计一个像 Youtube 这样的视频共享服务,用户可以上传/查看/搜索视频。类似服务:netflix.com、vimeo.com、dailymotion.com、veoh.com 难度级别:中等

1. 为什么选择 YouTube?

YouTube 是世界上最受欢迎的视频共享网站之一。该服务的用户可以上传、查看、分享、评价和举报视频以及对视频添加评论。

2. 系统的要求和目标

为了进行本练习,我们计划设计一个更简单的 Youtube 版本,并满足以下要求:

功能要求:

1. 用户应该能够上传视频。
2. 用户应该能够分享和观看视频。
3. 用户应该能够根据视频标题进行搜索。
4. 我们的服务应该能够记录视频的统计数据,例如喜欢/不喜欢、总观看次数、
ETC。
5. 用户应该能够添加和查看视频评论。

非功能性要求:

1. 系统可靠性高,上传的视频不丢失。
2. 系统应该是高可用的。一致性可能会受到影响(为了可用性);如果用户有一段时间没有看到视频,应该没问题。
3. 用户在观看视频时应该有实时的体验,不应该感到任何延迟。

不在范围内:视频推荐、最受欢迎的视频、频道、订阅、稍后观看、收藏夹等。

3. 容量估计和约束

假设我们有 15 亿总用户,其中 8 亿是日常活跃用户。如果平均而言,
用户每天观看 5 个视频,那么每秒的总视频观看次数将为:

$$800M * 5 / 86400 秒 \Rightarrow 46K \text{ 视频/秒}$$

假设我们的上传:观看比例为 1:200,即对于每个上传的视频,我们有 200 个视频被观看,即每秒上传 230 个视频。

$$46K / 200 \Rightarrow 230 \text{ 个视频/秒}$$

存储估算:假设每分钟有 500 小时的视频上传到 Youtube。如果平均一分钟的视频需要 50MB 的存储空间（视频需要以多种格式存储）,则一分钟上传的视频所需的总存储空间为：

$$500 \text{ 小时} * 60 \text{ 分钟} * 50\text{MB} \Rightarrow 1500 \text{ GB}/\text{分钟} (25 \text{ GB}/\text{秒})$$

这些数字是在忽略视频压缩和复制的情况下估计的,这将改变我们的估计。

带宽估算:每分钟上传 500 小时的视频,假设每个视频上传需要 10MB/分钟的带宽,那么每分钟我们将获得 300GB 的上传量。

$$500 \text{ 小时} * 60 \text{ 分钟} * 10\text{MB} \Rightarrow 300\text{GB}/\text{分钟} (5\text{GB}/\text{秒})$$

假设上传:观看比例为 1:200,我们需要 1TB/s 的传出带宽。

4. 系统API

我们可以使用 SOAP 或 REST API 来公开我们服务的功能。视频上传和搜索的API定义如下：

```
uploadVideo (api_dev_key,video_title,video_description,标签[],category_id,default_language,
录音_详细信息、视频_内容)
```

参数：

api_dev_key (string): 注册账户的API开发者密钥。除其他外,这将用于根据分配的配额限制用户。

video_title (字符串) :视频的标题。

vide_description (字符串) :视频的可选描述。

Tags (string[]):视频的可选标签。

category_id (字符串) :视频的类别,例如电影、歌曲、人物等。

default_language (字符串) :例如英语、普通话、印地语等。

Recording_details (字符串) :录制视频的位置。

video_contents (stream):要上传的视频。

返回: (字符串)

上传成功将返回 HTTP 202 (请求已接受)并且视频编码完成后

用户会收到电子邮件通知,其中包含访问视频的链接。我们还可以公开一个可查询的 API,让用户了解其上传视频的当前状态。

```
searchVideo(api_dev_key,search_query,user_location,maximum_videos_to_return,page_token)
```

参数：

api_dev_key (字符串) :我们服务的注册帐户的 API 开发人员密钥。

search_query (字符串) :包含搜索词的字符串。

user_location (字符串) :执行搜索的用户的可选位置。

Maximum_videos_to_return (number):一次请求返回的最大结果数。

page_token (字符串) :此标记将指定结果集中应返回的页面。

返回： (JSON)

包含有关与搜索查询匹配的视频资源列表的信息的 JSON。每个视频资源将包含视频标题、缩略图、视频创建日期和观看次数。

StreamVideo (api_dev_key,video_id,偏移量,编解码器,分辨率)

参数：

api_dev_key (字符串) :我们服务的注册帐户的 API 开发人员密钥。

video_id (字符串) :标识视频的字符串。

offset (数字) :我们应该能够从任何偏移量流式传输视频;该偏移量是从视频开始算起的时间（以秒为单位）。如果我们支持从多个设备播放/暂停视频,我们需要将偏移量存储在服务器上。这将使用户能够开始在任何

设备从他们停止的同一点开始。

编解码器 (字符串) 和分辨率 (字符串) :我们应该从客户端发送 API 中的编解码器和分辨率信息,以支持多个设备的播放/暂停。想象一下,您正在电视的 Netflix 应用程序上观看视频,暂停它,然后开始在手机的 Netflix 应用程序上观看。在这种情况下,您需要

编解码器和分辨率,因为这两种设备具有不同的分辨率并使用不同的编解码器。

返回：(流)

来自给定偏移量的媒体流 (视频块) 。

5. 高层设计

在高层次上,我们需要以下组件：

1.处理队列:每个上传的视频都会被推送到处理队列中以出队
稍后用于编码、缩略图生成和存储。

2.编码器:将每个上传的视频编码为多种格式。

3.缩略图生成器:为每个视频生成一些缩略图。

4.视频和缩略图存储:将视频和缩略图文件存储在某个分布式文件中
贮存。

5.用户数据库:存储用户的信息,例如姓名、电子邮件、地址等。

6.视频元数据存储:元数据数据库,用于存储有关视频的所有信息,例如
标题、文件在系统中的路径、上传用户、总观看次数、喜欢、不喜欢等。它还将用于存储所有视频评论。

Youtube的高层设计

6. 数据库架构

视频元数据存储-MySQL

视频元数据可以存储在 SQL 数据库中。每个视频应存储以下信息：

- 视频ID
- 标题
- 描述 · 尺寸
- 缩略图
- 上传者/用户 · 点赞
总数
- 不喜欢的总数
- 总浏览次数

对于每个视频评论，我们需要存储以下信息：

- 评论ID
- 视频ID
- 用户身份
- 评论
- 创造时间

用户数据存储-MySQL

- 用户ID、姓名、电子邮件、地址、年龄、注册详细信息等。

7. 详细组件设计

该服务的读取量很大,因此我们将专注于构建一个可以快速检索视频的系统。我们预计读:写比率为 200:1,这意味着每个视频上传都有 200 个视频观看次数。

视频将存储在哪里?视频可以存储在[HDFS](#)等分布式文件存储系统中或者[GlusterFS](#)。

我们应该如何有效地管理读取流量?我们应该将读取流量与写入流量分开。由于我们将拥有每个视频的多个副本,因此我们可以将读取流量分布在不同的服务器上。对于元数据,我们可以采用主从配置,其中写入将首先到达主服务器,然后应用于所有从服务器。这种配置可能会导致数据过时,例如,当添加新视频时,其元数据将首先插入主设备中,并且在将其应用于从设备之前,我们的从设备将无法看到它;因此它将向用户返回过时的结果。

这种陈旧性在我们的系统中可能是可以接受的,因为它的持续时间非常短暂,用户将能够在几毫秒后看到新视频。

缩略图将存储在哪里?缩略图会比视频多得多。如果我们假设

每个视频都会有五个缩略图,我们需要一个非常高效的存储系统来服务巨大的读取流量。在决定使用哪个存储系统来存储缩略图之前,需要考虑两个因素:

1. 缩略图是小文件,例如每个最大 5KB。
2. 与视频相比,缩略图的阅读流量会更大。用户一次只会观看一个视频,但他们可能会看到一个包含 20 个其他视频缩略图的页面。

让我们评估一下将所有缩略图存储在磁盘上的情况。鉴于我们有大量文件,我们必须对磁盘上的不同位置执行大量搜索才能读取这些文件。这是相当低效的并且会导致更高的延迟。

[太表](#)这里可能是一个合理的选择,因为它将多个文件组合成一个块来存储在磁盘上,并且在读取少量数据时非常有效。这两者都是我们服务的两个最重要的要求。将热缩略图保留在缓存中还有助于改善延迟,并且鉴于缩略图文件较小,我们可以轻松地将大量此类文件缓存在内存中。

视频上传:由于视频可能很大,如果上传时连接断开,我们应该支持从同一点恢复。

视频编码:新上传的视频存储在服务器上,并在处理队列中添加新任务,将视频编码为多种格式。所有编码完成后,上传者将收到通知,视频可供查看/共享。

Youtube的详细组件设计

8. 元数据分片

由于我们每天都有大量的新视频，并且读取负载非常高，因此，我们需要将数据分布到多台机器上，以便能够高效地执行读/写操作。我们有很多选择来分片我们的数据。让我们一一了解一下对这些数据进行分片的不同策略：

基于UserID的分片：我们可以尝试将特定用户的所有数据存储在一台服务器上。在存储时，我们可以将 UserID 传递给哈希函数，该函数会将用户映射到数据库服务器，我们将在其中存储该用户视频的所有元数据。在查询用户的视频时，我们可以要求哈希函数找到保存用户数据的服务器，然后从那里读取数据。要按标题搜索视频，我们必须查询所有服务器，每个服务器将返回一组视频。然后，中央服务器将汇总这些结果并对其进行排名，然后再将其返回给用户。

这种方法有几个问题：

1. 如果用户变得受欢迎怎么办？持有该用户的服务器上可能有很多查询；这可能会造成性能瓶颈。这也会影响我们服务的整体表现。
2. 随着时间的推移，与其他用户相比，某些用户最终可能会存储大量视频。维持一个不断增长的用户数据的均匀分布是相当棘手的。

为了从这些情况中恢复，我们必须重新分区/重新分布我们的数据，或者使用一致的散列来平衡服务器之间的负载。

基于 VideoID 的分片 :我们的哈希函数会将每个 VideoID 映射到一个随机服务器 , 我们将在其中存储该视频的元数据。为了查找用户的视频 , 我们将查询所有服务器 , 每个服务器将返回一组视频。集中式服务器将汇总这些结果并对其进行排名 , 然后再将其返回给用户。这种方法解决了我们的热门用户问题 , 但将其转移到了热门视频上。

我们可以通过在数据库服务器前面引入缓存来存储热门视频来进一步提高性能。

9. 视频重复数据删除

随着大量用户上传大量视频数据 , 我们的服务将不得不处理广泛的视频重复问题。重复视频通常在宽高比或编码方面有所不同 , 可能包含覆盖或附加边框 , 或者可能是较长原始视频的摘录。重复视频的激增可能会产生多个层面的影响 :

1. 数据存储 : 保留同一视频的多个副本可能会浪费存储空间。
2. 缓存 : 重复的视频会占用空间 , 导致缓存效率下降
可用于独特的内容。
3. 网络使用 : 重复的视频还会增加必须通过网络发送到网内缓存系统的数据量。
4. 能源消耗 : 更高的存储、低效的缓存和网络使用可能会导致
能源浪费。

对于最终用户来说 , 这些低效率将以重复的搜索结果、更长的视频启动时间和中断的流媒体形式体现。

对于我们的服务来说 , 重复数据删除在早期最有意义。当用户上传视频时与对其进行后处理以稍后查找重复视频相比。内联重复数据删除将为我们节省大量资源 , 可用于编码、传输和存储视频的重复副本。一旦任何用户启动

上传视频后 , 我们的服务可以运行视频匹配算法 (例如 ,[块匹配](#)、[相位相关性](#) , 等) 来查找重复项。如果我们已经有正
在上传的视频的副本 , 我们可以停止上传并使用现有副本 , 或者继续上传并使用新上传的视频 (如果质量更高) 。如果新
上传的视频是现有视频的子部分 , 反之亦然 , 我们可以智能地将视频分成更小的块 , 以便我们只上传现有视频的部分

丢失的。

10. 负载均衡

我们应该使用一致哈希将我们的缓存服务器之间 , 这也将有助于平衡缓存服务器之间的负载。由于我们将使用基于静态哈希的方案将视频映射到主机名 , 因此由于每个视频的受欢迎程度不同 , 可能会导致逻辑副本上的负载不均匀。例如 , 如果某个视频变得流行 , 则与该视频对应的逻辑副本将比其他服务器经历更多的流量。然后 , 逻辑副本的这些不均匀负载可以转化为相应物理服务器上的不均匀负载分布。要解决此问题 , 请将任何繁忙的服务器合二为一

location 可以将客户端重定向到同一缓存位置中不太繁忙的服务器。对于这种情况,我们可以使用动态 HTTP 重定向。

然而,使用重定向也有其缺点。首先,由于我们的服务尝试在本地进行负载平衡,因此如果接收重定向的主机无法提供视频,则会导致多次重定向。

此外,每次重定向都需要客户端发出额外的 HTTP 请求;它还会导致视频开始播放之前出现更高的延迟。此外,层间 (或跨数据中心)重定向会将客户端引导至远程缓存位置,因为较高层缓存仅存在于少数位置。

11. 缓存

为了服务全球分布的用户,我们的服务需要大规模的视频传输系统。我们的服务应该使用大量地理上分布的视频缓存服务器将其内容推近用户。我们需要制定一种策略,既能最大限度地提高用户性能,又能均匀地分配缓存服务器上的负载。

我们可以为元数据服务器引入缓存来缓存热数据库行。在访问数据库之前使用 Memcache 缓存数据和应用程序服务器可以快速检查缓存中是否有所需的行。最近最少使用 (LRU)对于我们的系统来说是一个合理的缓存驱逐策略。根据此策略,我们首先丢弃最近最少查看的行。

如何构建更加智能的缓存?如果我们遵循80-20规则,即视频每日阅读量的20%产生了80%的流量,这意味着某些视频非常受欢迎,以至于大多数人都会观看它们;由此可见,我们可以尝试缓存每日阅读量的 20% 的视频和元数据。

12. 内容分发网络 (CDN)

CDN 是一个由分布式服务器组成的系统,它根据用户的地理位置、网页的来源和内容交付服务器向用户交付 Web 内容。查看我们的[缓存](#)中的“CDN”部分章节。

我们的服务可以将热门视频移动到 CDN:

- CDN 在多个位置复制内容。视频更有可能更接近用户,并且通过更少的跳数,视频将从更友好的网络进行流式传输。
- CDN 机器大量使用缓存,并且大多可以在内存不足的情况下提供视频。

未由 CDN 缓存的不太受欢迎的视频 (每天 1-20 次观看)可以由我们位于各个数据中心的服务器提供服务。

13. 容错

我们应该使用[一致性哈希](#)用于在数据库服务器之间分发。一致性哈希不仅有助于更换失效的服务器,还有助于在服务器之间分配负载。

设计预先输入建议

让我们设计一个实时建议服务,它会在用户输入搜索文本时向他们推荐术语。类似服务:自动建议、预先输入搜索难度:中等

1.什么是预先输入建议?

预先输入建议使用户能够搜索已知且经常搜索的术语。当用户在搜索框中键入内容时,它会尝试根据用户输入的字符来预测查询,并给出完成查询的建议列表。预先输入建议可帮助用户阐明他们的想法

搜索查询更好。这不是为了加快搜索过程,而是为了指导用户并帮助他们构建搜索查询。

2. 系统的要求和目标

功能要求:当用户输入查询时,我们的服务应该建议以用户输入的内容开头的前 10 个术语。

非功能要求:建议应实时显示。用户应该能够在 200 毫秒内看到建议。

3. 基本系统设计和算法

我们要解决的问题是,我们需要存储大量“字符串”,以便用户可以使用任何前缀进行搜索。我们的服务将建议与给定前缀匹配的下一个术语。

例如,如果我们的数据库包含以下术语:cap、cat、captain 或 Capital,并且用户输入了“cap”,则我们的系统应建议“cap”、“captain”和“capital”。

由于我们必须以最小的延迟提供大量查询,因此我们需要提出一个方案

可以有效地存储我们的数据,以便可以快速查询。我们不能依赖某些数据库来做到这一点;我们需要以高效的数据结构将索引存储在内存中。

能够满足我们的目的的最合适的数据结构之一是 Trie (发音为 “try”)。trie 是一种树状数据结构,用于存储短语,其中每个节点存储短语的一个字符

顺序的方式。例如,如果我们需要在 trie 中存储 “cap,cat,caption,captain,capital” ,它看起来像:

现在,如果用户输入“cap”,我们的服务可以遍历trie 转到节点“P”以查找以此前缀开头的所有术语(例如,caption、capital 等)。

我们可以合并只有一个分支的节点以节省存储空间。上面的trie 可以这样存储:

我们应该有不区分大小写的 trie 吗?为了简单起见和搜索用例,我们假设我们的数据是 case 不敏感。

如何找到热门建议?既然我们可以找到给定前缀的所有术语,那么我们如何知道我们应该建议的前 10 个术语是什么?一个简单的解决方案可能是存储在每个节点终止的搜索计数,例如,如果用户搜索了“CAPTAIN”100 次和“CAPTION”500 次,我们可以将该数字与短语的最后一个字符一起存储。因此,现在如果用户输入“CAP”,我们就知道前缀“CAP”下搜索次数最多的单词是“CAPTION”。因此,给定一个前缀,我们可以遍历它下面的子树来找到最上面的建议。

给定一个前缀,遍历它的子树需要多少时间?考虑到我们需要索引的数据量,我们应该期待一棵巨大的树。即使遍历子树也会花费很长时间,例如

短语“系统设计面试问题”有 30 级深。由于我们有非常严格的延迟要求,因此我们确实需要提高解决方案的效率。

我们可以为每个节点存储最重要的建议吗?这肯定可以加快我们的搜索速度,但需要大量额外的存储空间。我们可以在每个节点存储前 10 条建议,然后将其返回给用户。我们必须承受存储容量的大幅增加才能达到所需的效率。

我们可以通过仅存储终端节点的引用而不是存储整个短语来优化存储。为了找到建议的术语,我们需要使用终端节点的父引用进行遍历。我们还需要存储每个参考的频率,以跟踪最重要的建议。

我们如何构建这个特里树?我们可以有效地自下而上地构建我们的特里树。每个父节点将递归调用所有子节点来计算其最热门建议及其计数。父节点将结合所有子节点的最佳建议来确定他们的最佳建议。

如何更新特里树?假设每天有 50 亿次搜索,这意味着每秒大约有 6 万次查询。如果我们尝试为每个查询更新 trie,这将非常消耗资源,这也会妨碍我们的读取请求。处理此问题的一种解决方案可能是在一定时间间隔后离线更新我们的字典树。

当新的查询出现时,我们可以记录它们并跟踪它们的频率。我们可以记录每个查询,也可以采样并记录每 1000 个查询。例如,如果我们不想显示搜索次数少于 1000 次的术语,则可以安全地记录每第 1000 个搜索的术语。

我们可以有一个[Map-Reduce \(MR\)](#)设置为定期(例如每小时)处理所有日志数据。

这些 MR 作业将计算过去一小时内所有搜索词的频率。然后我们可以更新我们的尝试使用这个新数据。我们可以获取 trie 的当前快照,并使用所有新术语及其频率对其进行更新。我们应该离线执行此操作,因为我们不希望读取查询被更新 trie 请求阻止。我们可以有两种选择:

1. 我们可以在每台服务器上制作一份trie副本,以便离线更新。完成后我们可以切换到开始使用它并丢弃旧的。
2. 另一种选择是我们可以为每个 trie 服务器进行主从配置。我们可以更新当主设备正在服务流量时,从设备。更新完成后,我们可以将奴隶设为我们的奴隶新主人。我们稍后可以更新旧的 master,然后它也可以开始提供流量。

我们如何更新预先输入建议的频率?由于我们存储每个节点的预先输入建议的频率,因此我们也需要更新它们。我们只能更新差异

频率而不是从头开始重新计算所有搜索词。如果我们要统计过去 10 天内搜索的所有术语,则需要从不再包含的时间段中减去计数,并添加包含的新时间段的计数。我们可以根据[指数移动平均线 \(EMA\)](#)添加和减去频率每个术语的。在 EMA 中,我们给予更多的权重

到最新数据。它也称为指数加权移动平均线。

在 trie 中插入一个新术语后,我们将转到该短语的终端节点并增加其频率。由于我们在每个节点中存储前 10 个查询,因此该特定搜索词可能会跳转到其他几个节点的前 10 个查询中。因此,我们需要更新这些节点的前 10 个查询。我们必须从节点一直遍历到根。对于每个父项,我们检查当前查询是否属于前 10 条。如果是,我们更新相应的频率。

如果不是,我们检查当前查询的频率是否足够高,可以进入前 10 个查询。如果是,我们插入这个新术语并删除频率最低的术语。

我们如何从 trie 中删除一个术语?假设由于某些法律问题或仇恨或盗版等原因,我们必须从 trie 中删除一个术语。当定期更新发生时,我们可以从 trie 中完全删除这些术语,同时,我们可以在每个服务器上添加一个过滤层,这将在将其发送给用户之前删除任何此类术语。

建议的不同排名标准可能是什么?除了简单的统计之外,对于术语排名,我们还必须考虑其他因素,例如新鲜度、用户位置、语言、人口统计、个人历史等。

4. Trie的永久存储

如何将 trie 存储在文件中,以便我们可以轻松地重建我们的 trie - 当机器重新启动时需要这样做?我们可以定期拍摄 trie 的快照并将其存储在文件中。这将使

如果服务器出现故障,我们可以重建一个字典树。存储时,我们可以从根节点开始,逐级保存trie。对于每个节点,我们可以存储它包含的字符以及它有多少个子节点。

在每个节点之后,我们应该放置它的所有子节点。假设我们有以下 trie:

如果我们将这个 trie 存储在具有上述方案的文件中,我们将有:“C2,A2,R1,T,P,O1,D”。

由此,我们可以轻松地重建我们的特里树。

如果您注意到了,我们不会存储每个节点的热门建议及其计数。存储这些信息很困难;由于我们的 trie 是自上而下存储的,因此我们没有在父节点之前创建子节点,因此没有简单的方法来存储它们的引用。为此,我们必须重新计算所有顶级术语的计数。这可以在我们构建特里树时完成。每个节点都会计算其最重要的建议并将其传递给其父节点。每个父节点将合并其所有子节点的结果,以找出其最重要的建议。

5. 规模估算

如果我们正在构建一个与 Google 规模相同的服务,我们预计每天会有 50 亿次搜索,这将为我们带来大约每秒 6 万次查询。

由于 50 亿条查询中会有大量重复项,我们可以假设其中只有 20% 会被重复。

独特的。如果我们只想对前 50% 的搜索词建立索引,我们可以去掉很多搜索频率较低的查询。假设我们有 1 亿个唯一术语要为其构建索引。

存储估计:如果平均每个查询由 3 个单词组成,并且单词的平均长度为 5 个字符,则平均查询大小将为 15 个字符。假设我们需要 2 个字节

存储一个字符,我们将需要 30 个字节来存储平均查询。所以我们需要的总存储空间:

$$1 \text{ 亿} * 30 \text{ 字节} \Rightarrow 3 \text{ GB}$$

我们可以预期这些数据每天都会有所增长,但我们也应该删除一些不再搜索的术语。如果我们假设每天有 2% 的新查询,并且如果我们在过去一年中维护索引,则我们应该预期的总存储量:

$$3\text{GB} + (0.02 * 3 \text{ GB} * 365 \text{ 天}) \Rightarrow 25 \text{ GB}$$

6. 数据分区

虽然我们的索引可以很容易地放在一台服务器上,但我们仍然可以对其进行分区,以满足我们更高效率和更低延迟的要求。我们如何有效地对数据进行分区以将其分布到多个服务器上?

A. 基于范围的分区:如果我们根据短语的首字母将短语存储在单独的分区中会怎样?因此,我们将以字母“A”开头的所有术语保存在一个分区中,将所有以字母“B”开头的术语保存到另一个分区中,依此类推。我们甚至可以组合某些不常出现的字母

到一个数据库分区。我们应该静态地提出这种分区方案,以便我们始终能够以可预测的方式存储和搜索术语。

这种方法的主要问题是,它可能会导致服务器不平衡,例如,如果我们决定

将所有以字母“E”开头的术语放入一个数据库分区中,但后来我们意识到以字母“E”开头的术语太多,无法放入一个数据库分区中。

我们可以看到,上述问题对于每个静态定义的方案都会发生。无法计算我们的每个分区是否静态地适合一台服务器。

b. 基于服务器的最大容量进行分区:假设我们根据服务器的最大内存容量对 trie 进行分区。只要服务器有可用内存,我们就可以继续将数据存储在服务器上。每当子树无法放入服务器时,我们就会破坏那里的分区,将该范围分配给该服务器,然后移动到下一个服务器以重复此过程。假设我们的第一个 trie 服务器可以存储从“A”到“AABC”的所有术语,这意味着我们的下一个服务器将从“AABD”开始存储。如果我们的第二个服务器最多可以存储“BXA”,则下一个服务器将从“BXB”开始,依此类推。我们可以保留一个哈希表来快速访问这个分区方案:

服务器 1,A-AABC

服务器2,AABD-BXA

服务器 3,BXB-CDA

对于查询,如果用户输入了“**A**”,我们必须查询服务器1和2以找到最重要的建议。

当用户输入“**AA**”时,我们仍然需要查询服务器1和2,但是当用户输入“**AAA**”时,我们只需要查询服务器1。

我们可以在trie服务器前面放置一个负载均衡器,它可以存储此映射并重定向流量。此外,如果我们从多个服务器进行查询,我们要么需要在服务器端合并结果以计算整体顶级结果,要么让我们的客户端这样做。如果我们更喜欢在服务器端执行此操作,

我们需要在负载均衡器和trie服务器之间引入另一层服务器(我们称之为聚合器)。这些服务器将聚合来自多个trie服务器的结果,并将最上面的结果返回给客户端。

基于最大容量的分区仍然可以将我们引导到热点,例如,如果有大量以“**cap**”开头的术语的查询,则持有它的服务器与其他服务器相比将具有较高的负载。

C. 基于术语的哈希进行分区:每个术语将被传递给哈希函数,该函数将生成一个服务器编号,我们将将该术语存储在该服务器上。这将使我们的术语分布

随机,从而最大限度地减少热点。要找到某个术语的预输入建议,我们必须询问所有服务器,然后汇总结果。

7. 缓存

我们应该意识到,缓存搜索最多的术语将对我们的服务非常有帮助。一小部分查询将承担大部分流量。我们可以在trie服务器前面设置单独的缓存服务器,保存最常搜索的术语及其预输入建议。应用程序服务器应该在访问trie服务之前检查这些缓存服务器,以查看它们是否具有所需的搜索项。

我们还可以构建一个简单的机器学习(ML)模型,该模型可以尝试根据简单的计数、个性化或趋势数据等来预测每个建议的参与度,并缓存这些术语。

8. 复制和负载均衡器

我们应该为我们的trie服务器提供副本,以实现负载平衡和容错。我们还需要一个负载均衡器来跟踪我们的数据分区方案并根据前缀重定向流量。

9. 容错

当trie服务器宕机时会发生什么?如上所述,我们可以有主从配置;如果主服务器死亡,从服务器可以在故障转移后接管。任何恢复的服务器都可以根据上次快照重建特里树。

10. 预先输入客户端

我们可以在客户端进行以下优化,以提升用户体验:

1. 如果用户在50毫秒内未按任何键,客户端应仅尝试访问服务器。

2. 如果用户不断打字,客户端可以取消正在进行的请求。
 3. 最初,客户端可以等待,直到用户输入几个字符。
 4. 客户端可以从服务器预取一些数据以保存将来的请求。
 5. 客户端可以在本地存储最近的建议历史记录。近代历史发生率非常高
被重用。
 6. 与服务器建立早期连接被证明是最重要的之一
因素。一旦用户打开搜索引擎网站,客户端就可以打开与服务器的连接。因此,当用户输入第一个字符时,客
户端不会浪费时间来建立连接。
7. 服务器可以将其缓存的某些部分推送到 CDN 和互联网服务提供商 (ISP) 以供使用
效率。

11. 个性化

用户将收到一些基于他们的历史搜索、位置、语言等的预先输入建议。我们可以将每个用户的个人历史记录分别存
储在服务器上并将其缓存在客户端上
也。服务器可以在将其发送给用户之前将这些个性化术语添加到最终集中。
个性化搜索应该始终优先于其他搜索。

设计 API 速率限制器

让我们设计一个 API 速率限制器,它将根据用户发送的请求数量来限制用户。难度级别:中等

1.什么是速率限制器?

想象一下,我们有一个服务正在接收大量请求,但它只能服务有限的服务
每秒的请求数。为了解决这个问题,我们需要某种节流或速率限制机制,只允许一定数量的请求,以便我们的服务可
以响应所有请求。速率限制器在较高级别上限制实体 (用户、设备、IP 等) 在特定时间窗口内可以执行的事件数量。
例如:

- 用户每秒只能发送一条消息。 · 每天只允许用户进
行三次失败的信用卡交易。 · 单个IP每天只能创建20个帐户。

一般来说,速率限制器会限制发送者在特定时间窗口内可以发出的请求数量。一旦达到上限,它就会阻止请求。

2. 为什么需要API限速？

速率限制有助于保护服务免受针对应用程序层的滥用行为，例如**拒绝服务 (DOS)**攻击、暴力密码尝试、暴力信用卡交易等。这些攻击通常是一系列 HTTP/S 请求，看起来像是来自真实用户，但通常是由机器（或机器人）生成的。因此，这些攻击通常更难检测，并且更容易导致服务、应用程序或 API 瘫痪。

速率限制还用于防止收入损失、降低基础设施成本、阻止垃圾邮件以及阻止在线骚扰。以下是可以通过使服务（或 API）更可靠而受益于速率限制的场景列表：

· 行为不当的客户端/脚本:无论是有意还是无意，某些实体可能会

发送大量请求使服务不堪重负。另一种情况可能是当用户发送大量低优先级请求时，我们希望确保它不会影响高优先级流量。例如，用户发送大量分析数据请求

不应妨碍其他用户的关键交易。

· 安全性:通过限制允许用户执行的第二因素尝试（在双因素身份验证中）的次数，例如，允许他们尝试使用错误密码的次数。

· 为了防止滥用行为和不良设计实践:如果没有 API 限制，客户端应用程序的开发人员将使用草率的开发策略，例如，一遍又一遍地请求相同的信息。

· 控制成本和资源使用:服务通常是针对正常输入行为而设计的，例如，用户在一分钟内写一篇文章。计算机可以通过 API 轻松地每秒推送数千次。速率限制器支持对服务 API 的控制。
· 收入:某些服务可能希望根据客户的级别来限制运营

服务，从而创建基于速率限制的收入模型。服务提供的所有 API 可能都有默认限制。为了超越这个限制，用户必须购买更高的限额。
· 为了消除流量高峰:确保该服务对其他人来说仍然可用。

3. 系统的要求和目标

我们的速率限制器应满足以下要求：

功能要求：

1. 限制实体在一个时间窗口内可以向 API 发送的请求数量，例如每秒 15 个请求。

2. API 是通过集群访问的，所以需要考虑不同服务器之间的速率限制。每当在单个服务器内或跨服务器组合超过定义的阈值时，用户应该收到错误消息。

非功能性要求：

1. 系统应该是高可用的。速率限制器应该始终有效,因为它可以保护我们的服务免受外部攻击。
2. 我们的速率限制器不应引入影响用户体验的大量延迟。

4. 如何进行限速?

速率限制是一个用于定义消费者访问 API 的速率和速度的过程。限制是在给定时间段内控制客户使用 API 的过程。可以在应用程序级别和/或 API 级别定义限制。当超过限制时,服务器返回 HTTP 状态“429 - 请求过多”。

5. 节流有哪些不同类型?

以下是不同服务使用的三种著名的限制类型:

硬限流: API请求数量不能超过限流限制。

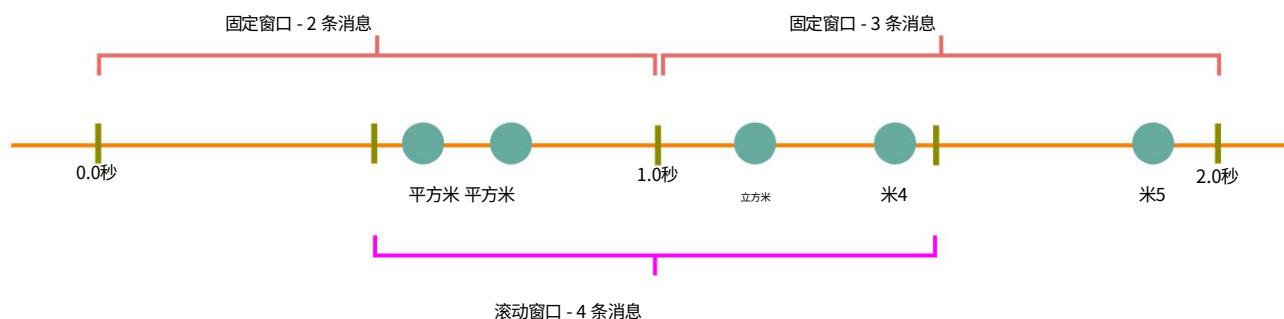
Soft Throttling: 在这种类型中,我们可以将API请求限制设置为超过一定的百分比。例如,如果我们的速率限制为每分钟 100 条消息并且超出限制 10%,则我们的速率限制器将允许每分钟最多 110 条消息。

弹性或动态限制: 在弹性限制下,如果系统有一些可用资源,请求数可能会超出阈值。例如,如果只允许用户每分钟发送100条消息,那么当系统中有空闲资源时,我们可以让用户每分钟发送100条以上的消息。

6. 用于速率限制的算法有哪些不同类型?

以下是用于速率限制的两种类型的算法:

固定窗口算法: 在该算法中,时间窗口被认为是从时间单元的开始到时间单元的结束。例如,无论发出 API 请求的时间范围如何,周期都将被视为 0-60 秒一分钟。在下图中,0-1 秒之间有两条消息,1-2 秒之间有 3 个消息。如果我们的速率限制为每秒两条消息,则该算法将仅限制“m5”。



滚动窗口算法:在此算法中,时间窗口是根据发出请求的时间加上时间窗口长度的分数来考虑的。例如,如果在一秒的第 300 毫秒和第 400 毫秒发送了两条消息,那么我们将从该秒的第 300 毫秒到下一秒的第 300 毫秒将它们计为两条消息。在上图中,每秒保留两条消息,我们将限制“m3”和“m4”。

7. 速率限制器的高级设计

速率限制器将负责决定 API 服务器将处理哪些请求以及将拒绝哪些请求。一旦新请求到达,Web 服务器首先要求速率限制器决定是对其进行服务还是对其进行限制。如果请求未被限制,那么它将被传递到 API

服务器。

速率限制器的高级设计

8. 基本系统设计和算法

让我们举个例子,我们要限制每个用户的请求数量。在这种情况下,对于每个唯一用户,我们将保留一个表示该用户发出了多少请求的计数以及开始对请求进行计数时的时间戳。我们可以将其保存在哈希表中,其中“键”将是“UserID”,“值”将是一个包含“Count”整数和“Count”整数的结构

大纪元时间:

核心价值

例如， 用户 ID { 计数, 开始时间 }</s
{3,1499818564}</

假设我们的速率限制器允许每个用户每分钟 3 个请求,因此每当有新请求进入时,我们的速率限制器将执行以下步骤:

1. 如果哈希表中不存在 “UserID” ,则将其插入,将 “Count”设置为 1,将 “StartTime”设置为当前时间 (标准化为一分钟) ,并允许请求。
2. 否则,查找 “UserID”的记录,如果 $\text{CurrentTime} - \text{StartTime} \geq 1$ 分钟,则将 “StartTime”设置为当前时间,“Count”设置为 1,并允许该请求。
3. 如果当前时间 - 开始时间 ≤ 1 分钟并且
 - 如果 “Count < 3” ,则增加Count 并允许请求。 · 如果 “Count ≥ 3 ” ,则拒绝请求。

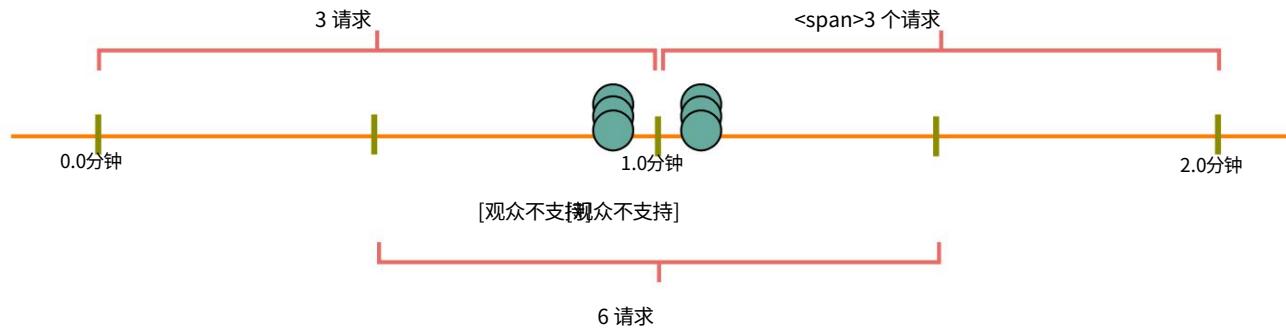
速率限制器允许用户“Kristie”每分钟发出三个请求



我们的算法存在哪些问题？

- 1.这是一个固定窗口算法,因为我们在每个结束时重置“StartTime”分钟,这意味着每分钟可能允许两倍的请求数量。想象一下,如果克里斯蒂在一分钟的最后一秒发送三个请求,那么她可以立即发送

在下一分钟的第一秒又发出了 3 个请求,导致两秒内出现了 6 个请求。这个问题的解决方案是滑动窗口算法,我们将在稍后讨论。



2.原子性:在分布式环境中,“读然后写”行为可能会产生竞争条件。想象一下,如果克里斯蒂当前的“计数”是“2”并且她又发出两个请求。如果两个单独的进程为每个请求提供服务,并在其中任何一个更新计数之前同时读取计数,则每个进程都会认为 Kristie 可能还有一个请求,并且她没有达到速率限制。



如果我们使用[Redis](#)为了存储我们的键值,解决原子性问题的一种解决方案是使用[Redis锁](#)在读更新操作期间。然而,这将以减慢同一用户的并发请求并引入另一层复杂性为代价。我们可以使用[Memcached](#),但它也会有类似的并发症。

如果我们使用简单的哈希表,我们可以有一个自定义实现来“锁定”每条记录来解决我们的原子性问题。

我们需要多少内存来存储所有用户数据?让我们假设一个简单的解决方案,将所有数据保存在哈希表中。

假设“UserID”占用 8 个字节。我们还假设 2 字节“计数”(最多可计数 65k)足以满足我们的用例。虽然纪元时间需要 4 个字节,但我们可以选择仅存储分钟和秒部分,这可以容纳 2 个字节。因此,我们总共需要 12 个字节来存储用户数据:

$$8 + 2 + 2 = 12 \text{ 字节}$$

假设哈希表的每条记录的开销为 20 字节。如果我们需要随时跟踪 100 万用户，则需要的总内存为 32MB：

$$(12 + 20) \text{ 字节} * 100 \text{ 万} => 32\text{MB}$$

如果我们假设需要一个 4 字节的数字来锁定每个用户的记录来解决原子性问题，则总共需要 36MB 内存。

这可以轻松地安装在单个服务器上；然而，我们不希望通过一台机器路由所有流量。此外，如果我们假设速率限制为每秒 10 个请求，那么对于我们的速率限制器来说，这将转化为 1000 万个 QPS！这对于单个服务器来说太多了。实际上，我们可以假设

我们会在分布式设置中使用 Redis 或 Memcached 等解决方案。我们将把所有数据存储在远程 Redis 服务器中，所有速率限制器服务器将在服务或限制任何请求之前读取（并更新）这些服务器。

9. 滑动窗口算法

如果我们能够跟踪每个用户的每个请求，我们就可以维护一个滑动窗口。我们可以将每个请求的时间戳存储在 Redis Sorted Set 中在哈希表的“值”字段中。

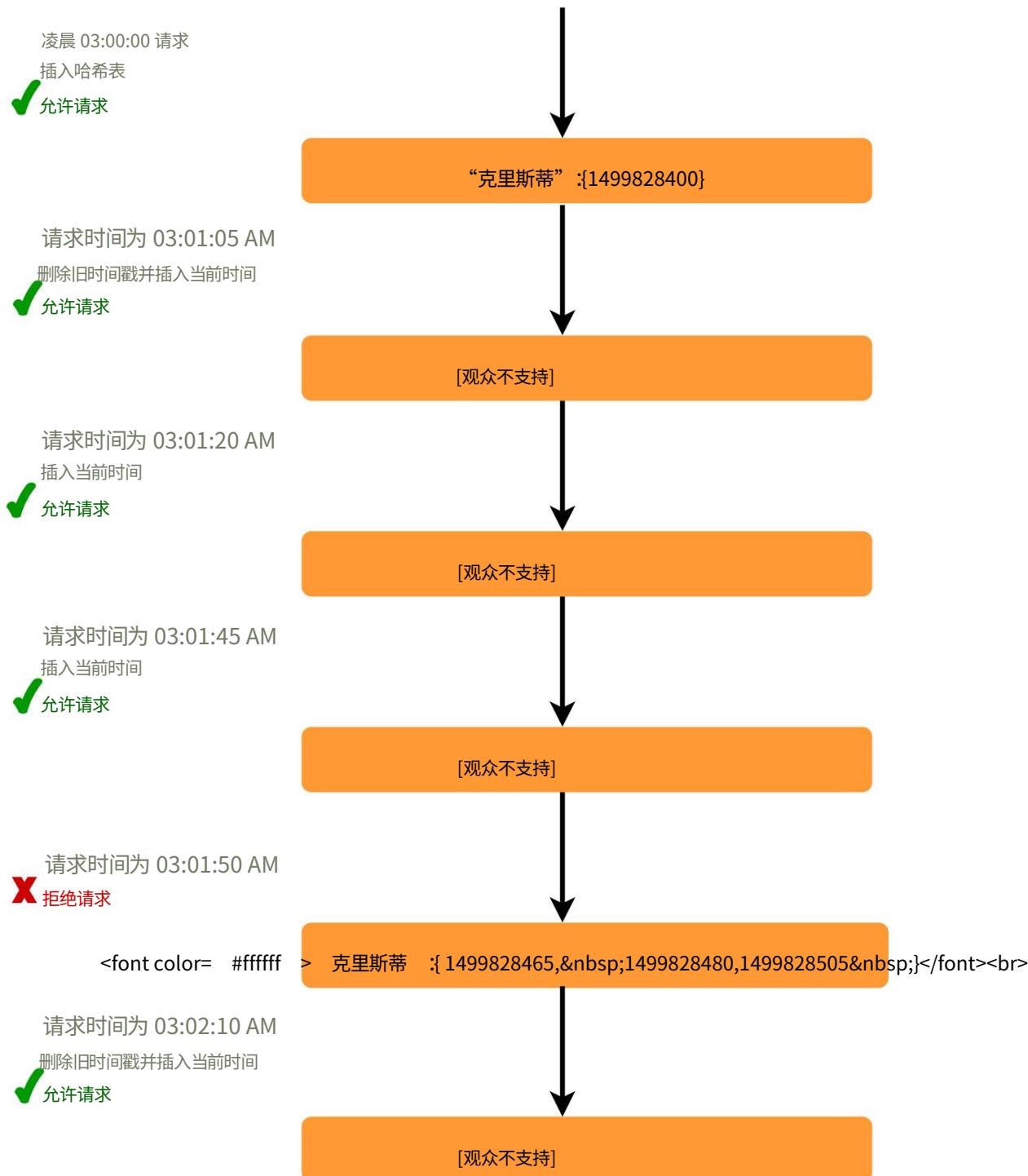
关键值

例如，
1499818000, 1499818500, 1499818860

假设我们的速率限制器允许每个用户每分钟三个请求，因此，每当有新请求时进来后，速率限制器将执行以下步骤：

1. 从排序集中删除所有早于 “`CurrentTime - 1 分钟`” 的时间戳。
2. 计算排序集中的元素总数。如果此计数更大，则拒绝请求超过我们的限制 “3”。
3. 将当前时间插入排序集中并接受请求。

速率限制器允许用户“Kristie”每分钟发出三个请求



我们需要多少内存来存储滑动窗口的所有用户数据？假设“UserID”占用 8 个字节。每个纪元时间需要 4 个字节。假设我们需要将速率限制为每小时 500 个请求。我们假设哈希表有 20 字节的开销，排序集有 20 字节的开销。最多，我们总共需要 12KB 来存储一个用户的数据：

$$8 + (4 + 20 \text{ (排序集开销)}) * 500 + 20 \text{ (哈希表开销)} = 12\text{KB}$$

这里我们为每个元素保留 20 字节的开销。在有序集合中,我们可以假设我们至少需要两个指针来维护元素之间的顺序 - 一个指向下一个元素的指针,一个指向下一个元素的指针。在 64 位机器上,每个指针将占用 8 个字节。所以我们需要 16 个字节作为指针。我们添加了一个额外的字 (4 个字节) 来存储其他开销。

如果我们需要随时跟踪 100 万用户,则需要的总内存为 12GB:

$$12\text{KB} * 100 \text{ 万} \approx 12\text{GB}$$

滑动窗口算法相比固定窗口占用大量内存,这将是一个可扩展性问题。如果我们可以结合上述两种算法来优化我们的内存使用呢?

10. 带计数器的滑动窗口

如果我们使用多个固定时间窗口 (例如速率限制时间窗口大小的 1/60) 跟踪每个用户的请求计数会怎样。例如,如果我们有每小时的速率限制,我们可以保留每分钟的计数,并在收到新请求以计算限制限制时计算过去一小时内所有计数器的总和。这将减少我们的内存占用。让我们举个例子

我们将速率限制为每小时 500 个请求,并附加限制为每分钟 10 个请求。这意味着当过去一小时内带有时间戳的计数器总和超过请求阈值 (500) 时,Kristie 已超出速率限制。除此之外,她每分钟发送的请求不能超过十个。这将是一个合理且实际的考虑,因为真正的用户不会发送频繁的请求。即使他们这样做了,他们也会通过重试看到成功,因为他们的限制每分钟都会重置。

我们可以将计数器存储在[Redis 哈希](#)中因为它为少于 100 个密钥提供了极其高效的存储。当每个请求增加哈希中的计数器时,它还会将哈希设置为一小时后过期。

我们将每个“时间”标准化为一分钟。

速率限制器允许用户“Kristie”每分钟发出三个请求



我们需要多少内存来存储带有计数器的滑动窗口的所有用户数据？

假设“UserID”占用 8 个字节。每个纪元时间需要 4 个字节，计数器需要 2 个字节。假设我们需要将速率限制为每小时 500 个请求。假设哈希表有 20 个字节的开销，Redis 哈希有 20 个字节的开销。由于我们将记录每分钟的计数，因此每个用户最多需要 60 个条目。我们总共需要 1.6KB 来存储一个用户的数据：

$$8 + (4 + 2 + 20 \text{ (Redis 哈希开销)}) * 60 + 20 \text{ (哈希表开销)} = 1.6\text{KB}$$

如果我们需要随时跟踪 100 万用户,则需要的总内存为 1.6GB:

$$1.6\text{KB} * 100 \text{ 万} \approx 1.6\text{GB}$$

因此,我们的“带计数器的滑动窗口”算法比简单的滑动窗口算法使用的内存少 86%。

11. 数据分片和缓存

我们可以根据“UserID”进行分片来分发用户的数据。对于容错和复制,我们应该使用一致性哈希。如果我们希望对不同的 API 有不同的限制,我们可以选择对每个 API 每个用户进行分片。以 URL 缩短器为例;我们可以为每个用户或 IP 的 createURL() 和 deleteURL() API 设置不同的速率限制器。

如果我们的 API 是分区的,那么实际的考虑可能是为每个 API 分片设置一个单独的(稍微小一些)速率限制器。让我们以 URL 缩短器为例,我们希望限制每个用户每小时创建的短 URL 不超过 100 个。假设我们对 createURL() API 使用基于哈希的分区,我们可以对每个分区进行速率限制,以允许用户除了每小时 100 个短 URL 之外,每分钟创建不超过 3 个短 URL。

我们的系统可以通过缓存最近的活跃用户获得巨大的好处。应用服务器可以在访问后端服务器之前快速检查缓存中是否有所需的记录。我们的速率限制器可以通过仅更新缓存中的所有计数器和时间戳来显着受益于回写缓存。可以以固定的时间间隔写入永久存储器。这样我们就可以确保速率限制器给用户请求添加的延迟最小。读取总是可以先命中缓存;一旦用户达到最大限制,并且速率限制器将仅读取数据而不进行任何更新,这将非常有用。

最近最少使用(LRU)对于我们的系统来说是一个合理的缓存驱逐策略。

12. 我们应该按IP还是按用户进行速率限制?

让我们讨论一下使用这些方案的优缺点:

IP:在此方案中,我们限制每个 IP 的请求;尽管它在区分“好”和“坏”参与者方面并不是最佳选择,但它仍然比完全没有速率限制要好。基于 IP 的限制的最大问题是当多个用户共享一个公共 IP(例如在网吧或使用同一网关的智能手机用户)时。一名不良用户可能会导致其他用户受到限制。

缓存基于 IP 的限制时可能会出现另一个问题,因为黑客甚至可以从一台计算机上获取大量 IPv6 地址,因此让服务器耗尽跟踪 IPv6 地址的内存是微不足道的!

用户:用户认证后可以对 API 进行限速。经过身份验证后,将向用户提供一个令牌,用户将在每次请求时传递该令牌。这将确保我们对具有有效身份验证令牌的特定 API 进行速率限制。但是如果我们必须限制速率怎么办

登录 API 本身?这种速率限制的弱点是,黑客可以通过输入错误的凭据来对用户执行拒绝服务攻击（达到限制）；之后实际用户将无法登录。

如果我们将以上两种方案结合起来呢？

混合:正确的方法可能是对每个 IP 和每个用户进行速率限制,因为它们在单独实现时都有弱点,但这将导致更多的缓存条目,每个条目有更多的详细信息,因此需要更多的内存和存储。

设计 Twitter 搜索

Twitter 是最大的社交网络服务之一,用户可以在其中分享照片、新闻和基于文本的消息。在本章中,我们将设计一个可以存储和搜索用户推文的服务。类似问题:推文搜索。难度级别:中等

1. 什么是 Twitter 搜索?

Twitter 用户可以随时更新他们的状态。每个状态（称为推文）均由纯文本组成
我们的目标是设计一个允许搜索所有用户推文的系统。

2. 系统的要求和目标

- 假设 Twitter 总用户数为 15 亿,其中每日活跃用户数为 8 亿。 · Twitter 平均每天收到 4 亿条推文。
- 一条推文的平均大小为 300 字节。 · 假设每
- 天有 5 亿次搜索。 · 搜索查询将由多个单词与 AND/OR 组合而成。

我们需要设计一个能够高效存储和查询推文的系统。

3. 容量估计和约束

存储容量:由于我们每天有 4 亿条新推文,平均每条推文为 300 字节,因此我们需要的总存储空间为:

$$400M * 300 \Rightarrow 120GB/\text{天}$$

每秒总存储量:

$$120GB / 24 \text{ 小时} / 3600 \text{ 秒} \approx 1.38MB/\text{秒}$$

4. 系统API

我们可以使用 SOAP 或 REST API 来公开我们服务的功能;以下是搜索 API 的定义:

搜索 (api_dev_key、search_terms、maximum_results_to_return、排序、page_token)

参数:

api_dev_key (string): 注册账户的API开发者密钥。除其他外,这将用于根据分配的配额限制用户。

search_terms (字符串) :包含搜索词的字符串。

Maximum_results_to_return (number):要返回的推文数量。

sort (数字) :可选排序模式:最新优先 (0 - 默认)、最匹配 (1)、最喜欢 (2)。

page_token (字符串) :此标记将指定结果集中应返回的页面。

返回: (JSON)

包含有关与搜索查询匹配的推文列表的信息的 JSON。每个结果条目可以包含用户 ID 和姓名、推文文本、推文 ID、创建时间、点赞数等。

5. 高层设计

在高层,我们需要将所有雕像存储在数据库中,并建立一个索引来跟踪哪个单词出现在哪条推文中。该索引将帮助我们快速找到用户试图搜索的推文。

Twitter 搜索的高级设计

6. 详细组件设计

1. 存储: 我们每天需要存储120GB的新数据。鉴于如此大量的数据,我们需要提出一种数据分区方案,将数据有效地分布到多个服务器上。如果我们规划未来五年,我们将需要以下存储:

$$120\text{GB} * 365\text{天} * 5\text{年} \approx 200\text{TB}$$

如果我们不想在任何时候都超过 80%,那么我们大约需要 250TB 的总存储空间。假设我们想要保留所有推文的额外副本以实现容错;那么,我们总共

存储需求为 500TB。如果我们假设一台现代服务器最多可以存储 4TB 的数据,那么我们将需要 125 台这样的服务器来保存未来五年所需的所有数据。

让我们从一个简单的设计开始,将推文存储在 MySQL 数据库中。我们可以假设

我们将推文存储在一个有两列的表中:TweetID 和 TweetText。假设我们根据 TweetID 对数据进行分区。如果我们的 TweetID 在系统范围内是唯一的,我们可以定义一个哈希函数

可以将 TweetID 映射到存储服务器,我们可以在其中存储该推文对象。

我们如何创建系统范围内唯一的 TweetID?如果我们每天收到 4 亿条新推文,那么五年后我们可以预期有多少个推文对象?

$$4\text{亿} * 365\text{天} * 5\text{年} \Rightarrow 7300\text{亿}$$

这意味着我们需要一个 5 字节的数字来唯一标识 TweetID。假设我们有一个服务,每当我们需要存储对象时,它都可以生成唯一的 TweetID (此处讨论的 TweetID 与设计 Twitter 中讨论的 TweetID 类似)。我们可以将 TweetID 提供给哈希函数来查找存储服务器并将我们的推文对象存储在那里。

2.索引:我们的索引应该是什么样的?由于我们的推文查询将由单词组成,因此让我们构建索引可以告诉我们哪个单词出现在哪个推文对象中。我们首先估计一下我们的索引有多大。如果我们想为所有英语单词和一些著名名词(如人名、城市名称等)建立一个索引,并且假设我们有大约 300K 个英语单词和 200K 个名词,那么我们的索引中总共将有 500k 个单词。指数。我们假设一个单词的平均长度是五个字符。如果我们将索引保存在内存中,则需要 2.5MB 内存来存储所有单词:

$$500K * 5 \Rightarrow 2.5 \text{ MB}$$

假设我们希望将过去两年内所有推文的索引保留在内存中。

由于我们将在 5 年内收到 730B 条推文,因此两年内我们将收到 292B 条推文。假设每个 TweetID 为 5 个字节,那么我们需要多少内存来存储所有 TweetID?

$$292B * 5 \Rightarrow 1460 \text{ GB}$$

因此,我们的索引就像一个大型分布式哈希表,其中“键”是单词,“值”是包含该单词的所有推文的 TweetID 列表。假设平均每条推文中有 40 个单词,并且由于我们不会索引介词和其他小词,如“the”、“an”、“and”等,所以我们假设每条推文中大约 15 个单词需要被索引。这意味着每个 TweetID 将在我们的索引中存储 15 次。因此,我们需要存储索引的总内存:

$$(1460 * 15) + 2.5\text{MB} \approx 21 \text{ TB}$$

假设高端服务器有 144GB 内存,我们需要 152 台这样的服务器来保存我们的索引。

我们可以根据两个标准对数据进行分片:

基于单词的分片 :在构建索引时 ,我们将迭代推文的所有单词并计算每个单词的哈希值以找到将对其进行索引的服务器。查找所有推文

如果包含特定单词 ,我们只需查询包含该单词的服务器。

这种方法有几个问题 :

1.如果某个词很火怎么办 ?然后服务器上就会有很多包含该单词的查询。

这种高负载会影响我们服务的性能。

2. 随着时间的推移 ,与其他单词相比 ,某些单词最终可能会存储大量 TweetID ,因此 ,在推文增长时保持单词的均匀分布非常棘手。

为了从这些情况中恢复 ,我们要么必须重新分区数据 ,要么使用一致性哈希。

基于推文对象的分片 :在存储时 ,我们将 TweetID 传递给哈希函数来查找服务器并索引该服务器上推文的所有单词。在查询特定单词时 ,

我们必须查询所有服务器 ,每个服务器都会返回一组 TweetID 。集中式服务器将聚合这些结果以将其返回给用户。

详细的组件设计

7. 容错

当索引服务器挂掉时会发生什么 ?我们可以拥有每台服务器的辅助副本 ,如果主服务器死亡 ,它可以在故障转移后接管控制权。主服务器和辅助服务器都将具有相同的索引副本。

如果主服务器和辅助服务器同时挂掉怎么办 ?我们必须分配一个新服务器并在其上重建相同的索引。我们怎样才能做到这一点 ?我们不知道该服务器上保存了哪些单词 / 推文。如果我们使用 “ 基于推文对象的分片 ” ,强力解决方案将是迭代整个数据库并使用我们的哈希函数过滤 TweetID ,以找出将存储在该服务器上的所有所需推文。这会是低效的 ,而且在这段时间内

当服务器被重建时,我们将无法从它提供任何查询,因此丢失了一些用户应该看到的推文。

我们如何有效地检索推文和索引服务器之间的映射?我们必须构建一个反向索引,将所有 TweetID 映射到其索引服务器。我们的索引生成器服务器可以保存这个信息。我们需要构建一个哈希表,其中“键”是索引服务器编号,“值”是包含该索引服务器上保存的所有 TweetID 的哈希集。请注意,我们是将所有 TweetID 保存在 HashSet 中;这将使我们能够快速从索引中添加/删除推文。因此,现在,每当索引服务器需要重建自身时,它都可以简单地向索引构建器服务器询问它需要存储的所有推文,然后获取这些推文来构建索引。这种方法肯定会很快。我们还应该有一个索引生成器服务器的副本以实现容错。

8. 缓存

为了处理热门推文,我们可以在数据库前面引入一个缓存。我们可以使用[Memcached](#),它可以将所有此类热门推文存储在内存中。应用程序服务器在访问后端数据库之前可以快速检查缓存中是否有该推文。根据客户的使用模式,我们可以调整我们需要的缓存服务器数量。对于缓存驱逐策略,最近最少使用 (LRU)似乎适合我们的系统。

9. 负载均衡

我们可以在系统中的两个位置添加负载平衡层:1)客户端和应用程序服务器之间以及2)应用程序服务器和后端服务器之间。最初,可以采用简单的循环方法;在后端服务器之间平均分配传入请求。这个LB是

实施简单并且不会引入任何开销。这种方法的另一个好处是负载均衡会将失效的服务器从轮换中剔除,并停止向其发送任何流量。循环负载均衡的一个问题是它不会考虑服务器负载。如果服务器过载或速度缓慢,负载均衡器不会停止向该服务器发送新请求。为了解决这个问题,可以放置更智能的负载均衡解决方案,定期查询后端服务器的负载情况,并据此调整流量。

10. 排名

如果我们想按社交图谱距离、流行度、相关性等对搜索结果进行排名怎么样?

假设我们想按受欢迎程度对推文进行排名,例如推文获得了多少点赞或评论,在这种情况下,我们的排名算法可以计算“受欢迎程度”(基于喜欢的数量等)并将其与索引一起存储。每个分区可以根据这个流行度数字对结果进行排序。在将结果返回到聚合器服务器之前,聚合器服务器组合所有这些结果,根据受欢迎程度对它们进行排序,并将排名靠前的结果发送给用户。

设计网络爬虫

让我们设计一个网络爬虫,它将系统地浏览和下载万维网。网络爬虫也称为网络蜘蛛、机器人、蠕虫、步行者和机器人。难度级别：困难

1.什么是网络爬虫？

网络爬虫是一种以有条理和自动化的方式浏览万维网的软件程序。它通过从一组起始页面递归获取链接来收集文档。

许多网站,特别是搜索引擎,使用网络爬行作为提供最新数据的手段。

搜索引擎下载所有页面并在其上创建索引以执行更快的搜索。

网络爬虫的其他一些用途是：

- 测试网页和链接的语法和结构是否有效。 · 监控站点以查看其结构或内容
- 何时发生变化。 · 维护流行网站的镜像站点。 · 搜索版权侵权行为。 · 建立一个特殊用途的索引,例如,对Web 上多媒体文件中存储的内容有一定了解的索引。

2. 系统的要求和目标

假设我们需要抓取整个网络。

可扩展性:我们的服务需要可扩展,以便它可以抓取整个 Web,并可用于获取数亿个 Web 文档。

可扩展性:我们的服务应该以模块化的方式设计,并期望添加新的功能。将来可能需要下载和处理更新的文档类型。

3. 一些设计考虑

爬行网络是一项复杂的任务,有很多方法可以完成它。我们应该问一些在继续之前的问题：

它是只针对 HTML 页面的爬虫吗?或者我们应该获取并存储其他类型的媒体,例如声音文件、图像、视频等?这很重要,因为答案可以改变设计。如果我们正在编写一个通用爬虫来下载不同的媒体类型,我们可能想要分解

将解析模块分成不同的模块集:一个用于 HTML,另一个用于图像,或者另一个用于视频,其中每个模块提取对该媒体类型感兴趣的内容。

现在我们假设我们的爬虫仅处理 HTML,但它应该是可扩展的,并且可以轻松添加对新媒体类型的支持。

我们正在关注哪些协议？HTTP？FTP 链接怎么样？我们的爬虫应该处理哪些不同的协议？为了练习的目的，我们将假设 HTTP。再说一遍，这不应该是以后很难扩展设计以使用 FTP 和其他协议。

我们预计要抓取的页面数量是多少？URL 数据库将会有多少？

假设我们需要抓取 10 亿个网站。由于网站可以包含很多很多 URL，因此我们假设我们的爬虫将访问的不同网页的上限为 150 亿个。

什么是“机器人排除”以及我们应该如何处理它？礼貌的网络爬虫实施机器人排除协议，该协议允许网站管理员声明其网站的某些部分禁止爬虫访问。机器人排除协议要求网络爬虫在从网站下载任何实际内容之前，先获取名为 robots.txt 的特殊文档，其中包含来自网站的这些声明。

4. 容量估计和约束

如果我们想在四个星期内抓取 150 亿个页面，那么每秒需要抓取多少个页面？

$$15B / (4 \text{ 周} * 7 \text{ 天} * 86400 \text{ 秒}) \approx 6200 \text{ 页/秒}$$

那么存储呢？页面大小差异很大，但是，如上所述，我们将仅处理 HTML 文本，我们假设平均页面大小为 100KB。对于每个页面，如果我们存储 500 字节的元数据，则需要的总存储空间为：

$$15B * (100KB + 500) \approx 1.5 \text{ PB}$$

假设 70% 容量模型（我们不想超过存储系统总容量的 70%），我们需要的总存储量：

$$1.5 \text{ 拍字节} / 0.7 \approx 2.14 \text{ 拍字节}$$

5. 高层设计

任何网络爬虫执行的基本算法都是以种子 URL 列表作为输入，并重复执行以下步骤。

1. 从未访问过的 URL 列表中选择一个 URL。
2. 确定其主机名的 IP 地址。
3. 与主机建立连接，下载相应文档。
4. 解析文档内容以查找新的 URL。
5. 将新 URL 添加到未访问 URL 列表中。
6. 处理下载的文档，例如存储它或索引其内容等。
7. 返回步骤 1

如何爬行?

广度优先还是深度优先?通常使用广度优先搜索 (BFS)。然而,在某些情况下也会使用深度优先搜索 (DFS),例如,如果您的爬虫已经与网站建立了连接,它可能只是 DFS 该网站内的所有 URL,以节省一些握手开销。

路径升序爬网:路径升序爬网可以帮助发现大量孤立的资源或在特定网站的常规爬网中找不到入站链接的资源。在这个方案中,爬虫会爬行到它想要爬行的每个 URL 中的每个路径。例如,当给定种子 URL <http://foo.com/a/b/page.html> 时,它将尝试抓取 /a/b/、/a/ 和

/。

实现高效网络爬虫的难点

Web 的两个重要特征使得 Web 爬行成为一项非常困难的任务:

1. 网页量大:网页量大意味着网络爬虫在任何时候只能下载一小部分网页,因此网络爬虫必须足够智能来确定下载的优先级。

2. 网页的变化率。当今动态世界的另一个问题是互联网上的网页变化非常频繁。因此,当爬网程序从站点下载最后一个页面时,该页面可能会发生更改,或者可能会向该站点添加新页面。

一个最低限度的爬虫至少需要以下组件:

1. URL frontier:存储要下载的 URL 列表,并确定首先抓取哪些 URL 的优先级。
2. HTTP Fetcher:从服务器检索网页。
3. 提取器:从 HTML 文档中提取链接。
4. 重复消除器:确保相同的内容不会被无意中提取两次。
5. 数据存储:存储检索到的页面、URL 和其他元数据。

6. 详细组件设计

假设我们的爬虫在一台服务器上运行，并且所有爬行都是由多个线程完成的，其中每个线程循环执行下载和处理文档所需的所有步骤。

此循环的第一步是从共享 URL 边界中删除用于下载的绝对 URL。

绝对 URL 以一个方案（例如“HTTP”）开始，该方案标识应该用于下载它的网络协议。我们可以以模块化的方式实现这些协议以实现可扩展性，因此

以后如果我们的爬虫需要支持更多的协议，也可以轻松完成。根据 URL 的方案，工作人员调用适当的协议模块来下载文档。下载后，文档被放入文档输入流（DIS）中。将文档放入 DIS

将使其他模块能够多次重新读取该文档。

一旦文档写入 DIS，工作线程就会调用重复数据删除测试来确定

该文档（与不同的 URL 关联）之前是否已被看到。如果是，则不再进一步处理该文档，并且工作线程会从边界中删除下一个 URL。

接下来，我们的爬虫需要处理下载的文档。每个文档可以有不同的 MIME 类型，如 HTML 页面、图像、视频等。我们可以以模块化的方式实现这些 MIME 方案

这样，以后如果我们的爬虫需要支持更多的类型，我们就可以轻松实现。根据下载文档的 MIME 类型，工作器调用与该 MIME 类型关联的每个处理模块的 process 方法。

此外，我们的 HTML 处理模块将从页面中提取所有链接。每个链接都会转换为绝对 URL，并根据用户提供的 URL 过滤器进行测试，以确定是否应该下载它。如果 URL 通过过滤器，工作器将执行 URL-seen 测试，检查该 URL 是否以前见过，即是否位于 URL 边界或已被下载。如果 URL 是新的，则会将其添加到边界。

让我们一一讨论这些组件,看看如何将它们分布到多台机器上:

1. URL 边界: URL 边界是包含所有剩余 URL 的数据结构。

被下载。我们可以通过从种子集中的页面开始对网络执行广度优先遍历来进行爬行。通过使用 FIFO 队列可以轻松实现此类遍历。

由于我们将要抓取大量 URL 列表,因此我们可以将 URL 边界分布到多个服务器中。假设每台服务器上都有多个工作线程执行爬网任务。

我们还假设我们的哈希函数将每个 URL 映射到负责抓取它的服务器。

在设计分布式 URL 边界时,必须牢记以下礼貌要求:

1. 我们的爬虫不应该通过下载大量页面来使服务器过载。
2. 我们不应该有多台机器连接一个Web服务器。

为了实现这种礼貌约束,我们的爬虫可以在每个服务器上拥有一组不同的 FIFO 子队列。每个工作线程都有其单独的子队列,从该子队列中删除用于爬行的 URL。当需要添加新的 URL 时,其所在的 FIFO 子队列将由该 URL 的规范主机名确定。我们的哈希函数可以将每个主机名映射到一个线程

数字。这两点一起意味着,最多一个工作线程将从给定的 Web 服务器下载文档,并且通过使用 FIFO 队列,它不会使 Web 服务器过载。

我们的 URL 边界有多大?其大小将达到数亿个 URL。因此,我们需要将 URL 存储在磁盘上。我们可以以这样的方式实现队列,即它们具有单独的缓冲区用于入队和出队。入队缓冲区一旦填满,将转储到磁盘,而出队缓冲区将保留需要访问的 URL 的缓存;它可以定期读取

磁盘来填充缓冲区。

2. fetcher 模块: fetcher 模块的目的是使用适当的网络协议 (如 HTTP) 下载与给定 URL 相对应的文档。如上所述,网站管理员创建 robots.txt 来禁止爬虫访问其网站的某些部分。为了避免在每次请求时都下载此文件,我们的爬虫程序的 HTTP 协议模块可以维护一个固定大小的缓存,将主机名映射到机器人的排除规则。

3. 文档输入流: 我们的爬虫的设计使得同一个文档可以被多个处理模块处理。为了避免多次下载文档,我们使用称为文档输入流 (DIS) 的抽象在本地缓存文档。

DIS 是一个输入流,用于缓存从互联网读取的文档的全部内容。它还提供了重新阅读文档的方法。DIS 可以将小文档 (64 KB 或更少) 完全缓存在内存中,而较大的文档可以临时写入备份文件。

每个工作线程都有一个关联的 DIS,它在文档之间重复使用该 DIS。从边界提取 URL 后,工作人员将该 URL 传递到相关协议模块,该模块通过网络连接初始化 DIS 以包含文档的内容。那时的工人

将 DIS 传递给所有相关的处理模块。

4. 文档重复数据删除测试: Web 上的许多文档都可以通过多个不同的 URL 访问。

文档镜像到各个服务器上的情况也很多。这两种影响都会导致任何网络爬虫多次下载同一文档。为了防止多次处理文档,我们对每个文档执行重复数据删除测试以删除重复项。

为了执行此测试,我们可以计算每个已处理文档的 64 位校验和并将其存储在数据库中。对于每个新文档,我们可以将其校验和与所有先前计算的校验和进行比较,以查看该文档以前是否被见过。我们可以使用 MD5 或 SHA 来计算这些

校验和。

校验和存储有多大?如果我们的校验和存储的全部目的是进行重复数据删除,那么我们只需要保留一个包含所有先前处理的文档的校验和的唯一集合。

考虑到 150 亿个不同的网页,我们需要:

$$15B * 8 \text{ 字节} \Rightarrow 120 \text{ GB}$$

虽然这可以适应现代服务器的内存,但如果我们没有足够的可用内存,我们可以在每台服务器上保留较小的基于 LRU 的缓存,并由持久存储支持所有内容。

重复数据删除测试首先检查缓存中是否存在校验和。如果不是，则必须检查校验和是否驻留在后台存储器中。如果找到校验和，我们将忽略该文档。

否则，它将被添加到缓存和后台存储中。

5. URL 过滤器：URL 过滤机制提供了一种可定制的方式来控制下载的 URL 集。这用于将网站列入黑名单，以便我们的爬虫可以忽略它们。在将每个 URL 添加到边界之前，工作线程会查阅用户提供的 URL 过滤器。我们可以定义

过滤器按域、前缀或协议类型限制 URL。

6. 域名解析：在联系 Web 服务器之前，Web 爬虫必须使用域名服务（DNS）将 Web 服务器的主机名映射为 IP 地址。考虑到我们将使用的 URL 数量，DNS 名称解析将成为爬虫的一大瓶颈。为了避免重复请求，我们可以通过构建本地 DNS 服务器来开始缓存 DNS 结果。

7. URL 重复数据删除测试：在提取链接时，任何网络爬虫都会遇到指向同一地址的多个链接文档。为了避免多次下载和处理文档，必须先对每个提取的链接执行 URL 重复数据删除测试，然后再将其添加到 URL 边界。

为了执行 URL 重复数据删除测试，我们可以将爬虫程序看到的所有 URL 以规范形式存储在数据库中。为了节省空间，我们不存储 URL 集中每个 URL 的文本表示，而是存储固定大小的校验和。

为了减少数据库存储上的操作数量，我们可以在所有线程共享的每个主机上保留流行 URL 的内存缓存。之所以有这个缓存，是因为某些 URL 的链接非常常见，因此将流行的 URL 缓存在内存中会导致内存中的命中率很高。

URL 存储需要多少存储空间？如果校验和的全部目的是进行 URL 重复数据删除，那么我们只需要保留一个包含所有先前看到的 URL 校验和的唯一集合。考虑到 150 亿个不同的 URL 和 4 个字节的校验和，我们需要：

$$15B * 4 \text{ 字节} \Rightarrow 60 \text{ GB}$$

我们可以使用布隆过滤器进行重复数据删除吗？布隆过滤器是一种用于集合成员资格测试的概率数据结构，可能会产生误报。一个大的位向量代表该集合。通过计算元素的“n”个哈希函数并设置相应的位，将元素添加到集合中。如果设置了元素的所有“n”个散列位置处的位，则该元素被视为在集合中。因此，文档可能会被错误地视为在集合中，但不可能出现误报。

使用布隆过滤器进行 URL 可见测试的缺点是，每个误报都会导致 URL 不被添加到边界，因此，文档将永远不会被下载。机会

可以通过增大位向量来减少误报。

8. 检查点：整个网络的抓取需要数周时间才能完成。为了防止故障，我们的爬虫可以将其状态的定期快照写入磁盘。中断或中止的爬网很容易被从最新的检查点重新启动。

7.容错性

我们应该使用一致的哈希在爬行服务器之间进行分发。一致的哈希不仅有助于替换失效的主机,还有助于在爬行服务器之间分配负载。

我们所有的爬网服务器都将执行定期检查点并将其 FIFO 队列存储到磁盘。如果服务器出现故障,我们可以更换它。同时,一致的散列应该将负载转移到其他服务器。

8. 数据分区

我们的爬虫将处理三种数据:1) 要访问的 URL 2) 用于重复数据删除的 URL 校验和 3) 记录重复数据删除的校验和。

由于我们根据主机名分发 URL,因此可以将这些数据存储在同一主机上。因此,每个主机将存储其需要访问的一组 URL、所有先前访问过的 URL 的校验和以及所有下载文档的校验和。由于我们将使用一致的散列,因此我们可以假设 URL 将从过载的主机重新分发。

每个主机将定期执行检查点并将其持有的所有数据的快照转储到远程服务器上。这将确保如果一台服务器宕机,另一台服务器可以通过从最后一个快照获取其数据来替换它。

9. 履带式陷阱

有许多爬虫陷阱、垃圾邮件站点和隐藏内容。爬网程序陷阱是导致爬网程序无限期爬行的一个 URL 或一组 URL。有些履带式陷阱是无意的。例如,文件系统内的符号链接可以创建循环。其他爬虫陷阱是故意引入的。

例如,人们编写了动态生成无限文档网络的陷阱。此类陷阱背后的动机各不相同。反垃圾邮件陷阱旨在捕获垃圾邮件发送者用于寻找电子邮件地址的爬虫,而其他网站则使用陷阱来捕获搜索引擎爬虫以提高其搜索评级。

设计 Facebook 的新闻源

让我们设计 Facebook 的 Newsfeed,其中包含来自用户关注的所有人员和页面的帖子、照片、视频和状态更新。类似服务:Twitter 新闻源、Instagram 新闻源、Quora 新闻源 难度级别:难

1. Facebook 的新闻源是什么?

新闻源是 Facebook 主页中间不断更新的故事列表。它包括状态更新、照片、视频、链接、应用程序活动以及用户在 Facebook 上关注的人员、页面和群组的“点赞”。换句话说,它是您朋友和您的生活故事的完整可滚动版本的汇编,其中包括照片、视频、位置、状态更新和其他活动。

对于您设计的任何社交媒体网站 - Twitter、Instagram 或 Facebook - 您将需要一些新闻源系统来显示来自朋友和关注者的更新。

2. 系统的要求和目标

让我们为 Facebook 设计一个新闻源,满足以下要求:

功能要求:

1. 新闻源将根据用户发布的人员、页面和群组的帖子生成
接下来。
2. 一个用户可能有很多朋友并关注大量的页面/群组。
3. 源可能包含图像、视频或仅文本。
4. 我们的服务应该支持在新帖子到达所有活动的新闻源时添加新帖子
用户。

非功能性需求:

- 1.我们的系统应该能够实时生成任何用户的新闻源 - 最大延迟
最终用户的时间为 2 秒。
2. 假设有新的新闻源请求,则帖子到达用户源的时间不应超过 5 秒
进来。

3. 容量估计和约束

假设用户平均有 300 个朋友并关注 200 个页面。

流量估算:假设每日活跃用户数为 3 亿,每个用户平均每天获取时间线五次。这将导致每天 1.5B 个新闻源请求,即大约 17,500 个
每秒请求数。

存储估计:平均而言,假设每个用户的提要中需要有大约 500 个帖子,我们希望将其保留在内存中以便快速获取。我们还假设平均每个帖子的大小为 1KB。这意味着我们需要为每个用户存储大约 500KB 的数据。为了存储这一切

对于所有活跃用户的数据,我们需要 150TB 的内存。如果服务器可以容纳 100GB,我们将需要大约 1500 台机器来为所有活跃用户保留前 500 个帖子。

4. 系统API

一旦我们最终确定了需求,定义系统 API 总是一个好主意。

这应该明确说明系统的期望。

我们可以使用 SOAP 或 REST API 来公开我们服务的功能。以下是获取新闻源的 API 的定义:

getUserFeed(api_dev_key,user_id,since_id,count,max_id,exclusive_replies)

参数:

api_dev_key (字符串) :注册的 API 开发人员密钥可用于以下用途:

根据分配的配额限制用户。

user_id (数字) :系统将为其生成新闻源的用户的 ID。

since_id (数字) :可选;返回 ID 高于 (即比指定 ID 更新)的结果。

count (数量) :可选;指定要尝试检索的提要项目数,每个不同请求最多可达 200 个。

max_id (数字) :可选;返回 ID 小于 (即早于)或等于指定 ID 的结果。

except_replies(boolean):可选;此参数将阻止回复出现在返回的时间线中。

返回: (JSON) 返回包含提要项目列表的 JSON 对象。

5. 数据库设计

存在三个主要对象:用户、实体 (例如页面、组等) 和 FeedItem (或 Post)。以下是有关这些实体之间关系的一些观察:

- 用户可以关注其他实体并可以与其他用户成为朋友。
- 用户和实体都可以发布包含文本、图像或视频的 FeedItem。 · 每个 FeedItem 将有一个 UserID, 该 ID 将指向创建它的用户。为简单起见, 我们假设只有用户可以创建提要项目, 尽管在 Facebook 页面上也可以发布提要项目。
- 每个 FeedItem 都可以有一个 EntityID, 指向该页面或组。
帖子已创建。

如果我们使用关系数据库, 则需要对两种关系进行建模: 用户-实体关系和 FeedItem-媒体关系。由于每个用户可以与很多人成为朋友并关注很多实体, 因此我们可以将这种关系存储在单独的表中。“UserFollow”中的“Type”列标识所关注的实体是用户还是实体。类似地, 我们可以有一个 FeedMedia 关系表。

6. 高层系统设计

从高层次来看 ,这个问题可以分为两个部分：

Feed 生成： Newsfeed 是根据用户和实体（页面和实体）的帖子（或 Feed 项）生成的
用户关注的组）。因此,每当我们的系统收到为用户（例如 Jane)生成提要的请求时,我们将执行以下步骤：

1. 检索 Jane 关注的所有用户和实体的 ID。
2. 检索这些 ID 的最新、最受欢迎和相关的帖子。这些是潜在的职位
我们可以在 Jane 的新闻源中显示。
3. 根据与 Jane 的相关性对这些帖子进行排名。这代表 Jane 的当前提要。
4. 将此提要存储在缓存中，并返回要在 Jane 的提要上呈现的热门帖子（例如 20 个）。
5. 在前端,当 Jane 到达当前提要的末尾时,她可以获取接下来的 20 个帖子
来自服务器等。

这里需要注意的一件事是我们生成了一次 feed 并将其存储在缓存中。 Jane 关注的人发来的新帖子怎么样?如果Jane在线,我们应该有一个排名机制

并将这些新帖子添加到她的提要中。我们可以定期（比如每五分钟）执行上述步骤来排名并将新帖子添加到她的提要中。然后 Jane 会收到通知，其中有更新的项目
她可以获取的饲料。

提要发布:每当 Jane 加载新闻提要页面时，她都必须从服务器请求并提取提要项目。当她到达当前提要的末尾时，她可以从服务器提取更多数据。对于较新的项目，服务器可以通知 Jane，然后她可以拉取或服务器可以推送这些新帖子。我们稍后将详细讨论这些选项。

概括地说，我们的 Newsfeed 服务中需要以下组件：

1. **Web服务器**:维持与用户的连接。该连接将用于传输用户和服务器之间的数据。
2. **应用服务器**:执行在数据库服务器中存储新帖子的工作流程。我们还需要一些应用程序服务器来检索新闻源并将其推送给最终用户。
3. **元数据数据库和缓存**:存储有关用户、页面和组的元数据。
4. **帖子数据库和缓存**:存储有关帖子及其内容的元数据。
5. **视频和照片存储以及缓存**: Blob 存储，用于存储帖子中包含的所有媒体。
6. **新闻源生成服务**:收集用户的所有相关帖子并对其进行排名，以生成新闻源并存储在缓存中。该服务还将接收实时更新，并将这些较新的提要项目添加到任何用户的时间线中。
7. **Feed通知服务**:通知用户有更新的商品可供使用新闻源。

以下是我们系统的高级架构图。用户B和C正在关注用户A。

Facebook 新闻源架构

7. 详细组件设计

让我们详细讨论系统的不同组件。

A。饲料生成

让我们以新闻源生成服务为例,该服务从 Jane 关注的所有用户和实体获取最新帖子;查询将如下所示:

```
(从 FeedItem WHERE UserID in ( 中选择 FeedItemID  
    从 UserFollow 中选择 EntityOrFriendID,其中 UserID = <current_user_id> 且类型 = 0(用户))  
)  
联盟  
(从 FeedItem WHERE EntityID in ( 中选择 FeedItemID  
    SELECT EntityOrFriendID FROM UserFollow WHERE UserID = <current_user_id> and type = 1(entity))  
)  
按创建日期 DESC 排序  
限制 100
```

以下是 Feed 生成服务的设计存在的问题:

1. 对于有很多朋友/关注的用户来说速度太慢了,因为我们必须执行对大量帖子进行排序/合并/排名。
2. 当用户加载页面时,我们生成时间线。这会很慢并且有一个高延迟。
3. 对于实时更新,每次状态更新都会导致所有关注者的 feed 更新。这可能会导致我们的新闻源生成服务出现大量积压。
4. 对于实时更新,服务器向用户推送(或通知)新帖子可能会导致非常重的负载,特别是对于拥有大量关注者的人员或页面。为了提高效率,我们可以预生成时间线并将其存储在内存中。

离线生成新闻源:我们可以拥有专用服务器,不断生成用户的新闻源并将其存储在内存中。因此,每当用户请求为其提要提供新帖子时,我们都可以简单地从预生成的存储位置提供该帖子。使用此方案,用户的新闻源不会在加载时编译,而是定期编译,并在用户请求时返回给用户。

每当这些服务器需要为用户生成提要时,它们将首先查询以查看上次为该用户生成提要的时间。然后,从那时起将生成新的提要数据。我们可以将这些数据存储在哈希表中,其中“键”是 UserID,“值”是

像这样的结构:

```
结构体{
    LinkedHashMap<FeedItemID, FeedItem> feedItems;
    最后生成的日期时间;
}
```

我们可以将 FeedItemID 存储在类似于[Linked Hash Map](#)的数据结构中或[树形图](#),这不仅可以让我们跳转到任何提要项,还可以轻松地迭代地图。每当用户想要获取更多提要项目时,他们可以发送他们当前在新闻提要中看到的最后一个 FeedItemID,然后我们可以跳转到哈希映射中的该 FeedItemID 并从那里返回下一批/页面的提要项目。

我们应该在内存中为用户的 feed 存储多少个 feed 项?最初,我们可以决定为每个用户存储 500 个提要项目,但稍后可以根据使用模式调整此数字。例如,如果我们假设用户的 Feed 的一页有 20 个帖子,并且大多数用户从未浏览过超过 10 页的 Feed,则我们可以决定为每个用户仅存储 200 个帖子。对于任何想要查看更多帖子(超过内存中存储的帖子)的用户,我们始终可以查询后端服务器。

我们应该为所有用户生成(并保存在内存中)新闻源吗?会有很多用户不经常登录。我们可以采取以下一些措施来解决这个问题; 1)更直接的方法可能是使用基于 LRU 的缓存,该缓存可以将长时间未访问其新闻源的用户从内存中删除 2)更智能的解决方案可以找出用户的登录模式以预生成其新闻源新闻源,例如,用户在一天中的什么时间处于活动状态以及用户在一周期中的哪几天访问其新闻源? ETC。

现在让我们在下一节中讨论“实时更新”问题的一些解决方案。

b. 提要发布

将帖子推送给所有关注者的过程称为扇出。以此类推,推送方式称为“写入时扇出”,而拉动方式称为“加载时扇出”。让我们讨论向用户发布提要数据的不同选项。

1. “拉”模型或负载扇出:此方法涉及将所有最近的馈送数据保留在内存,以便用户可以在需要时从服务器中提取它。客户可以提取提要定期或在需要时手动获取数据。这种方法可能存在的问题
a) 在用户发出拉取请求之前,新数据可能不会显示给用户,b) 很难找到正确的拉取节奏,因为大多数情况下,如果没有新数据,拉取请求将导致空响应,造成资源浪费。

2. “推送”模型或写入时扇出:对于推送系统,一旦用户发布了帖子,我们可以立即将该帖子推送给所有关注者。优点是,在获取提要时,您无需浏览朋友的列表并获取每个人的提要。它显着减少了读取操作。为了有效地处理这个问题,用户必须维护**长轮询**向服务器请求接收更新。这种方法可能存在的问题是,当用户拥有数百万粉丝(名人用户)时,服务器必须将更新推送给很多人。

3. 混合:处理 Feed 数据的另一种方法可能是使用混合方法,即执行

写入时扇出和负载时扇出的组合。具体来说,我们可以停止推送来自拥有大量关注者(名人用户)的用户的帖子,而只推送那些拥有数百(或数千)关注者的用户的数据。对于名人用户,我们可以让关注者拉取更新。由于对于拥有大量朋友或关注者的用户来说,推送操作的成本可能非常高,因此通过为他们禁用扇出,我们可以节省大量资源。

另一种替代方法可能是,一旦用户发布帖子,我们就可以将扇出限制为仅限她的在线朋友。此外,为了从这两种方法中获益,“推送通知”和“拉取服务”最终用户的结合是一个很好的方法。纯粹的推或拉模型

通用性较差。

在每个请求中我们可以向客户返回多少个 Feed 项目?我们应该有一个最大的限制

用户可以在一次请求中获取的项目数(例如 20)。但是,我们应该让客户端指定每个请求需要多少个提要项目,因为用户可能希望根据设备(移动设备与桌面设备)获取不同数量的帖子。

如果他们的新闻源有新帖子,我们是否应该始终通知用户?当有新数据可用时,用户收到通知可能很有用。然而,在移动设备上,数据

使用成本相对较高,会消耗不必要的带宽。因此,至少对于移动设备,我们可以选择不推送数据,而是让用户“拉动刷新”来获取新的帖子。

8. Feed 排名

在新闻源中对帖子进行排名的最直接方法是根据帖子的创建时间,但当今的排名算法所做的远不止于此,以确保“重要”帖子排名更高。

排名的高级思想是首先选择使帖子变得重要的关键“信号”，然后找出如何将它们组合起来计算最终排名分数。

更具体地说，我们可以选择与任何提要项的重要性相关的特征，例如，点赞数、评论数、分享数、更新时间、帖子是否有图像/视频等，然后，分数可以使用这些特征进行计算。对于简单的排名系统来说，这通常足够了。一个更好的排名系统可以通过不断评估我们在用户粘性、保留率、广告收入等方面是否取得进展来显着改进自身。

9. 数据分区

A. 分片帖子和元数据

由于我们每天都有大量的新帖子，并且读取负载也非常高，因此我们需要将数据分发到多台机器上，以便我们可以高效地读取/写入数据。为了分片我们的存储帖子及其元数据的数据库，我们可以采用与[设计 Twitter 中讨论的类似设计](#)。

b. 分片提要数据

对于存储在内存中的 feed 数据，我们可以根据 UserID 对其进行分区。我们可以尝试存储一台服务器上的所有的用户的全部数据。存储时，我们可以将 UserID 传递给我们的哈希函数，该函数将用户映射到缓存服务器，我们将在其中存储用户的提要对象。此外，对于任何给定用户，由于我们不希望存储超过 500 个 FeedItemID，因此我们不会遇到用户的提要数据不适合单个服务器的情况。为了获取用户的提要，我们始终只需查询一台服务器。为了未来的增长和复制，我们必须使用[一致性哈希](#)。

设计 Yelp 或附近的朋友

让我们设计一个类似 Yelp 的服务，用户可以搜索附近的地点，如餐馆、剧院或购物中心等，还可以添加/查看地点的评论。
类似服务：邻近服务器。

难度级别：困难

1. 为什么选择 Yelp 或 Proximity Server？

邻近服务器用于发现附近的景点，例如地点、活动等。如果您以前没有使用过yelp.com，请在继续之前尝试一下（您可以搜索附近的餐馆、剧院等）并花一些时间了解不同的内容该网站提供的选项。这将有助于您更好地理解本章。

2. 系统的要求和目标

我们希望通过类似 Yelp 的服务实现什么目标？我们的服务将存储有关不同地点的信息，以便用户可以对其进行搜索。查询后，我们的服务将返回一个列表用户周围的地方。

我们的类似 Yelp 的服务应满足以下要求：

功能要求：

1. 用户应该能够添加/删除/更新地点。
2. 给定位置（经度/纬度），用户应该能够找到某个范围内所有附近的地点
给定半径。
3. 用户应该能够添加有关某个地点的反馈/评论。反馈可以有图片，
文本和评级。

非功能性要求：

1. 用户应该拥有最小延迟的实时搜索体验。
2. 我们的服务应该支持繁重的搜索负载。与添加新地点相比，将会有许多搜索请求。

3. 规模估算

让我们假设我们有 500M 位置和每秒 100K 查询 (QPS)，来构建我们的系统。我们还假设名额和 QPS 每年增长 20%。

4. 数据库架构

每个位置可以有以下字段：

1. LocationID (8字节) :唯一标识一个位置。
2. 名称 (256字节)
3. 纬度 (8字节)
4. 经度 (8字节)
5. 描述 (512字节)
6. 类别 (1字节) :例如咖啡店、餐厅、剧院等。

虽然 4 字节的数字可以唯一标识 500M 的位置，但考虑到未来的发展，我们将使用 8 字节的 LocationID。

总大小: $8 + 256 + 8 + 8 + 512 + 1 \Rightarrow 793$ 字节

我们还需要存储某个地点的评论、照片和评级。我们可以有一个单独的表来存储对地点的评论：

1. LocationID (8字节)
2. ReviewID (4字节) :唯一标识一条评论，假设任何位置不会有更多
超过 2^{32} 条评论。
3. 评论文本 (512字节)
4. 评级 (1字节) :一个地方获得多少颗星 (满分 10 颗星)。

同样，我们可以有一个单独的表来存储地点和评论的照片。

5. 系统API

我们可以使用 SOAP 或 REST API 来公开我们服务的功能。以下是搜索 API 的定义：

搜索 (api_dev_key、search_terms、user_location、radius_filter、maximum_results_to_return、category_filter、排序、page_token)

参数：

api_dev_key (string): 注册账户的API开发者密钥。除其他外，这将用于根据分配的配额限制用户。

search_terms (字符串) : 包含搜索词的字符串。

user_location (字符串) : 执行搜索的用户的位置。

radius_filter (number) : 可选搜索半径 (以米为单位)。

Maximum_results_to_return (number) : 要返回的业务结果数。

category_filter (字符串) : 用于过滤搜索结果的可选类别，例如餐厅、购物中心、ETC。

sort (数字) : 可选排序模式：最佳匹配 (0 - 默认)、最小距离 (1)、最高评分 (2)。

page_token (字符串) : 此标记将指定结果集中应返回的页面。

返回： (JSON)

包含有关与搜索查询匹配的企业列表的信息的 JSON。每个结果条目都将包含企业名称、地址、类别、评级和缩略图。

6. 基本系统设计和算法

在较高层面上，我们需要存储并索引上述每个数据集（地点、评论等）。为了让用户查询这个庞大的数据库，索引应该高效读取，因为在搜索附近的地方时，用户希望实时看到结果。

鉴于地点的位置不会经常改变，我们不需要担心数据的频繁更新。相比之下，如果我们打算构建一个对象经常改变其位置的服务，例如人或出租车，那么我们可能会提出一个非常不同的设计。

让我们看看存储这些数据的不同方法，并找出哪种方法最适合我们的情况

用例：

A。 SQL解决方案

一种简单的解决方案是将所有数据存储在 MySQL 等数据库中。每个地点都将存储在单独的行中，由 LocationID 唯一标识。每个地点都会将其经度和纬度分别存储在两个不同的列中，并进行快速搜索；我们应该在这两个字段上都有索引。

要查找给定位置 (X,Y) 在半径 “D” 内的所有附近地点，我们可以这样查询：

从纬度位于 XD 和 $X+D$ 之间且经度位于 YD 和 $Y+D$ 之间的地点中选择 *

上面的查询并不完全准确,因为我们知道要找到两点之间的距离,我们必须使用距离公式 (毕达哥拉斯定理) ,但为了简单起见,我们采用这个。

这个查询的效率如何?我们估计我们的服务中有 5 亿个存储位置。自从
我们有两个单独的索引,每个索引都可以返回一个巨大的位置列表,并且在这两个列表上执行交
集效率不高。看待这个问题的另一种方式是可以

“ $X-D$ ”和 “ $X+D$ ”之间的位置过多, “ $Y-D$ ”和 “ $Y+D$ ”之间的位置也类似。如果我们能以某种方式缩短这些
列表,就可以提高查询的性能。

b.网格

我们可以将整个地图划分为更小的网格,以将位置分组为更小的集合。每个网格将存储位于特定经度和
纬度范围内的所有地点。该计划将使

我们只查询几个网格来找到附近的地方。根据给定的位置和半径,我们可以找到所有
邻近的网格,然后查询这些网格以找到附近的地方。

我们假设 GridID (一个四字节数字) 将唯一标识我们系统中的网格。

合理的网格大小是多少?网格大小可以等于我们想要查询的距离,因为我们还想减少网格的数量。如果
网格大小等于我们想要的距离

来查询,那么我们只需要在包含给定位置和邻近的网格内搜索
八个网格。由于我们的网格是静态定义的 (根据固定的网格大小) ,我们可以轻松找到
任何位置 (纬度、经度) 及其相邻网格的网格号。

在数据库中,我们可以存储每个位置的 GridID,并在其上建立索引,以便更快地搜索。现在,我们的查询将如下所示:

从纬度介于 XD 和 $X+D$ 之间、经度介于 YD 和 $Y+D$ 之间的地点中选择 * ,

GridID 在 (GridID, GridID1, GridID2, ..., GridID8)

这无疑会提高我们查询的运行时间。

我们应该将索引保存在内存中吗?在内存中维护索引将提高我们服务的性能。我们可以将索引保存在哈希表中,其中“键”是网格编号,“值”是该网格中包含的位置列表。

我们需要多少内存来存储索引?假设我们的搜索半径是 10 英里;考虑到地球总面积约为 2 亿平方英里,我们将拥有 2000 万个网格。

我们需要一个四字节的数字来唯一标识每个网格,并且由于 LocationID 是 8 个字节,我们需要 4GB 内存 (忽略哈希表开销) 来存储索引。

$$(4 * 20M) + (8 * 500M) \approx 4 \text{ GB}$$

对于那些具有很多位置的网格,该解决方案仍然运行缓慢,因为我们的位置在网格之间分布不均匀。我们可以有一个有很多地方的密集区域,另一方面,我们可以有人口稀疏的区域。

如果我们可以动态调整网格大小,那么只要我们有一个包含很多位置的网格,我们就可以将其分解以创建更小的网格,这个问题就可以解决。这种方法面临的一些挑战可能是:1)如何将这些网格映射到位置;2)如何找到网格的所有相邻网格。

C. 动态尺寸网格

假设我们不希望网格中的位置超过 500 个,这样我们就可以进行更快的搜索。因此,每当一个网格达到这个限制时,我们就把它分成四个大小相等的网格,并在它们之间分配位置。这意味着人口稠密的地区 (例如旧金山市中心) 将有很多网格,而人口稀少的地区 (例如太平洋) 将有大型网格,且仅在沿海地区。

什么数据结构可以保存这些信息?每个节点有四个子节点的树可以

服务于我们的目的。每个节点将代表一个网格,并将包含有关该网格中所有位置的信息。如果一个节点达到 500 个位置的限制,我们将分解它以在其下创建四个子节点并在它们之间分配位置。这样,所有的叶子节点就代表了无法进一步细分的网格。因此叶节点将保留一个位置列表。这种每个节点可以有四个子节点的树结构称为[四叉树](#)



我们将如何构建四叉树?我们将从一个节点开始,该节点将在一个网格中代表整个世界。由于它将有超过 500 个位置,因此我们将其分解为四个节点并在它们之间分配位置。我们将对每个子节点不断重复此过程,直到没有剩余位置超过 500 个的节点。

我们如何找到给定位置的网格?我们将从根节点开始,向下搜索以找到所需的节点/网格。在每一步中,我们都会查看当前访问的节点是否有子节点。如果有,我们将移动到包含我们所需位置的子节点并重复此过程。如果该节点没有任何子节点,那么这就是我们想要的节点。

我们如何找到给定网格的相邻网格?由于只有叶节点包含位置列表,我们可以用双向链表连接所有叶节点。这样我们就可以在相邻叶节点之间向前或向后迭代以找出我们想要的位置。查找相邻网格的另一种方法是通过父节点。我们可以在每个节点中保留一个指针来访问其父节点,并且由于每个父节点都有指向其所有子节点的指针,因此我们可以轻松找到节点的兄弟节点。我们可以通过向上遍历父指针来继续扩展对相邻网格的搜索。

一旦我们有了附近的 LocationID,我们就可以查询后端数据库来查找有关这些地点的详细信息。

搜索工作流程是什么?我们将首先找到包含用户位置的节点。如果说节点有足够的位置,我们可以将它们返回给用户。如果没有,我们将继续扩展到相邻节点(通过父指针或双向链表),直到找到所需数量的位置或根据最大半径耗尽搜索。

存储四叉树需要多少内存?对于每个地点,如果我们仅缓存 LocationID 和纬度/经度,则需要 12GB 来存储所有地点。

$$24 * 500M \Rightarrow 12 \text{ GB}$$

由于每个网格最多可以有 500 个位置,而我们有 500M 个位置,那么总共有多少个网格我们将有?

$$500M / 500 \Rightarrow 1M \text{ 网格}$$

这意味着我们将有 1M 个叶节点,它们将保存 12GB 的位置数据。具有 1M 个叶节点的四叉树将具有大约 1/3 的内部节点,每个内部节点将有 4 个指针 (用于其子节点)。如果每个指针有 8 个字节,那么我们需要存储所有内部节点的内存为:

$$1M * 1/3 * 4 * 8 = 10 \text{ MB}$$

因此,保存整个四叉树所需的总内存将为 12.01GB。这可以很容易地适应现代服务器。

我们如何将一个新地点插入到我们的系统中?每当用户添加新地点时,我们都需要将其插入数据库和四叉树中。如果我们的树驻留在一台服务器上,那就很容易

添加一个新的地点,但如果四叉树分布在不同的服务器之间,首先我们需要找到新地点的网格/服务器,然后将其添加到那里 (在下一节中讨论)。

7. 数据分区

如果我们有大量的位置,以至于我们的索引无法放入单个机器的内存中怎么办?以每年 20% 的增长,未来我们将达到服务器的内存极限。另外,如果一台服务器无法提供所需的读取流量怎么办?为了解决这些问题,我们必须对四叉树进行分区!

我们将在这里探索两种解决方案 (这两种分区方案也可以应用于数据库) :

A. 基于区域的分片:我们可以将我们的地点划分为区域 (如邮政编码),这样属于一个区域的所有地点都将存储在固定节点上。为了存储一个地点,我们将通过其区域找到服务器,类似地,在查询附近的地点时,我们将询问包含用户位置的区域服务器。这种方法有几个问题:

1. 如果某个区域变得很热怎么办?持有该区域的服务器上会有很多查询,使其执行缓慢。这将影响我们的服务表现。
2. 随着时间的推移,某些区域最终可能会比其他区域存储更多的空间。因此,在区域不断扩大的同时保持地方的均匀分布是相当困难的。

为了从这些情况中恢复,我们必须重新分区数据或使用一致性哈希。

b. 基于 LocationID 的分片:我们的哈希函数会将每个 LocationID 映射到我们将存储该位置的服务器。在构建我们的四叉树时,我们将迭代所有位置并计算每个 LocationID 的哈希值以找到存储它的服务器。要查找某个位置附近的地点,

我们必须查询所有服务器,每个服务器将返回一组附近的地点。中央服务器将汇总这些结果并将其返回给用户。

我们在不同的分区上会有不同的四叉树结构吗?是的,这种情况可能会发生,因为不能保证所有分区上的任何给定网格中的位置数量相同。

但是,我们确实确保所有服务器具有大致相同数量的位置。不过,不同服务器上的这种不同的树结构不会导致任何问题,因为我们将搜索所有分区上给定半径内的所有相邻网格。

本章的剩余部分假设我们已经根据 LocationID 对数据进行了分区。

8. 复制和容错

拥有四叉树服务器的副本可以提供数据分区的替代方案。为了分配读取流量,我们可以拥有每个四叉树服务器的副本。我们可以采用主从配置,其中副本(从属)仅提供读取流量;所有写入流量将首先到达主站,然后应用于从站。从站可能没有一些最近插入的位置(会有几毫秒的延迟),但这可以接受。

当四叉树服务器死机时会发生什么?我们可以拥有每台服务器的辅助副本,如果主节点死亡,它可以在故障转移后接管控制权。主服务器和辅助服务器都将具有相同的四叉树结构。

如果主服务器和辅助服务器同时挂掉怎么办?我们必须分配一台新服务器并在其上重建相同的四叉树。既然我们不知道该服务器上保存了哪些位置,我们该如何做到这一点?强力解决方案是迭代整个数据库并使用我们的哈希函数过滤 LocationID,以找出将存储在该数据库中的所有所需位置。

服务器。这会是低效且缓慢的;另外,在服务器重建期间,我们将无法从服务器上提供任何查询,从而错过了一些应该由服务器看到的地方
用户。

我们如何有效地检索 Places 和 QuadTree 服务器之间的映射?我们必须建立一个反向索引,它将所有地点映射到其四叉树服务器。我们可以有一个单独的四叉树索引服务器来保存此信息。我们需要构建一个 HashMap,其中“键”是四叉树服务器编号,“值”是包含该四叉树服务器上保存的所有位置的哈希集。我们需要存储每个地点的 LocationID 和纬度/经度,因为信息服务器可以通过它构建他们的四叉树。请注意,我们将 Places 的数据保存在 HashSet 中,这将使我们能够快速从索引中添加/删除 Places。因此,现在,每当四叉树服务器需要重建自身时,它都可以简单地向四叉树索引服务器询问它需要存储的所有位置。

这种方法肯定会非常快。我们还应该有一个四叉树索引服务器的副本以实现容错。如果四叉树索引服务器挂掉,它始终可以通过迭代数据库来重建索引。

9. 缓存

为了处理热点,我们可以在数据库前面引入一个缓存。我们可以使用像 Memcache 这样的现成解决方案,它可以存储有关热点的所有数据。应用服务器在访问后端数据库之前,可以快速检查缓存中是否有该位置。根据客户的使用模式,我们可以调整我们需要多少个缓存服务器。对于缓存驱逐策略,最近最少使用 (LRU)似乎适合我们的系统。

10. 负载均衡 (LB)

我们可以在系统中的两个位置添加 LB 层:1) 在客户端和应用程序服务器之间以及 2) 应用程序服务器和后端服务器之间。最初,可以采用简单的循环方法;这将在后端服务器之间平均分配所有传入请求。这个 LB 很简单实施并且不会引入任何开销。这种方法的另一个好处是,如果服务器死机,负载均衡器会将其从轮换中删除,并停止向其发送任何流量。

循环负载均衡的一个问题是,它不会考虑服务器负载。如果服务器过载或速度缓慢,负载均衡器将不会停止向该服务器发送新请求。为了处理这个问题,一个

需要更智能的 LB 解决方案来定期查询后端服务器的负载并据此调整流量。

11. 排名

如果我们不仅想根据邻近度还想根据流行度或相关性对搜索结果进行排名,该怎么办?

我们如何返回给定半径内最受欢迎的地点?假设我们跟踪每个地方的整体受欢迎程度。聚合数字可以代表我们系统中的受欢迎程度,例如,一个地方在十颗星中获得多少颗星(这将是用户给出的不同排名的平均值)?我们将把这个数字存储在数据库和四叉树中。在搜索给定半径内的前 100 个位置时,我们可以要求四叉树的每个分区返回前 100 个位置。

最受欢迎的地方。然后聚合器服务器可以确定不同分区返回的所有位置中的前100个位置。

请记住,我们构建的系统并不是为了经常更新地点的数据。通过这种设计,我们如何修改四叉树中某个地点的受欢迎程度?虽然我们可以在四叉树中搜索一个地点并更新其受欢迎程度,但这会占用大量资源,并且会影响搜索请求和系统吞吐量。假设一个地方的受欢迎程度预计不会在几个小时内反映在系统中,我们可以决定每天更新一次或两次,特别是当系统负载最小时。

我们的下一个问题,设计 Uber 后端,详细讨论了四叉树的动态更新。

设计 Uber 后端

让我们设计一个像 Uber 这样的乘车共享服务,将需要乘车的乘客与拥有汽车的司机联系起来。类似服务:Lyft、滴滴、Via、Sidecar 等。 难度级别:硬 先决条件:设计 Yelp

1. 优步是什么?

优步允许其客户预订司机乘坐出租车。优步司机使用他们的私家车载客出行。顾客和司机都通过智能手机相互沟通使用优步应用程序。

2. 系统的要求和目标

让我们从构建一个更简单的 Uber 版本开始。

我们的系统中有两种用户:1)司机2)顾客。

- 驾驶员需要定期向服务部门通报其当前位置以及是否可以接听电话
- 挑选乘客。 · 乘
- 客可以看到附近所有可用的司机。 · 客户可以叫车;附近的司机
- 会收到有关顾客已准备好接载的通知
- 向上。
- 一旦司机和顾客接受乘车,他们就可以不断地看到彼此的当前位置
- 直到旅程结束。 · 到达
- 目的地后,司机将旅程标记为完成,以便可以搭乘
- 下一次骑行。

3. 容量估计和约束

- 假设我们有 3 亿客户和 100 万司机,以及 100 万每日活跃客户,
- 50 万日活跃司机。

- 假设每天有 100 万次骑行。 ·
- 假设所有活跃驾驶员每三秒通知一次其当前位置。 · 一旦客户提出乘车请求,系统应该能够实时联系司机
- 时间。

4. 基本系统设计和算法

我们将采用[设计 Yelp](#)中讨论的解决方案并修改它以使其适用于上述“Uber”用例。我们最大的区别是,我们的四叉树在构建时并没有考虑到它会频繁更新。因此,我们的动态网格解决方案存在两个问题:

- 由于所有活跃驾驶员每三秒报告一次他们的位置,因此我们需要更新我们的数据结构来反映这一点。如果我们必须为驱动程序的每次更改更新四叉树的位置,需要花费大量的时间和资源。要将驱动程序更新到新位置,我们必须根据驱动程序之前的位置找到正确的网格。如果新职位不属于当前网格,我们必须从当前网格中删除驱动程序并将用户移动/重新插入到正确的网格。此举之后,如果新网格达到驱动程序的最大限制,我们必须重新对其进行分区。 · 我们需要有一个快速机制,将所有附近司机的当前位置传播给该区域的任何活跃客户。此外,当乘车正在进行时,我们的系统需要通知驾驶员和乘客汽车的当前位置。

尽管我们的四叉树可以帮助我们快速找到附近的驱动程序,但不能保证树中的快速更新。

每次司机报告其位置时,我们是否都需要修改四叉树?如果我们不使用驱动程序的每次更新来更新四叉树,它将包含一些旧数据,并且不会正确反映驱动程序的当前位置。如果您还记得的话,我们构建四叉树的目的是有效地找到附近的驱动程序(或地点)。由于所有活跃的驱动程序每三秒报告一次他们的位置,因此我们的树上发生的更新比查询附近的驱动程序要多得多。那么,如果我们将所有驱动程序报告的最新位置保留在哈希表中并稍微降低更新四叉树的频率呢?假设我们保证驾驶员的当前位置将在 15 秒内反映在四叉树中。同时,我们将维护一个哈希表,用于存储驾驶员报告的当前位置;我们将此称为 DriverLocationHT。

DriverLocationHT 需要多少内存?我们需要将 DriveID、它们当前和旧的位置存储在哈希表中。因此,我们总共需要 35 个字节来存储一条记录:

- 1.DriverID (3字节-100万个驱动程序)
- 2.旧纬度 (8字节)
- 3.旧经度 (8字节)
- 4.新纬度 (8字节)
5. 新经度 (8 字节)总计 = 35 字节

如果我们总共有 100 万个驱动程序,我们需要以下内存(忽略哈希表开销):

$$100 \text{ 万} * 35 \text{ 字节} \Rightarrow 35 \text{ MB}$$

我们的服务将消耗多少带宽来接收所有驾驶员的位置更新?要是我们
获取 DriverID 及其位置,它将是 (3+16 => 19 字节)。如果我们每三个收到此信息
从 100 万个驱动程序开始,我们每三秒将获得 19MB 的数据。

我们需要将 DriverLocationHT 分发到多个服务器上吗?尽管我们的内存和带宽要求不需要这样做,因为所有这些信
息都可以轻松存储在一台服务器上,但是,为了可扩展性、性能和容错性,我们应该将 DriverLocationHT 分发到多台服务器
上。我们可以根据DriverID进行分发,使分发完全随机。

我们将持有 DriverLocationHT 的机器称为驱动程序位置服务器。除了存储驾驶员的位置之外,每个服务器还将执行两
件事:

1. 一旦服务器收到驾驶员位置的更新,他们就会将该信息广播给所有感兴趣的客户。
2. 服务器需要通知各自的QuadTree服务器刷新驱动程序的位置。作为
如上所述,这种情况每 10 秒就会发生一次。

如何高效地将司机的位置广播给客户?我们可以有一个推送模型
服务器将位置推送给所有相关用户。我们可以有一个专门的通知服务,可以向所有感兴趣的客户广播司机的当前位置。我们
可以建造

我们的通知服务采用发布者/订阅者模型。当客户在其手机上打开 Uber 应用程序时
他们通过手机查询服务器以查找附近的司机。在服务器端,在将驱动程序列表返回给客户之前,我们将向客户订阅这些驱动
程序的所有更新。我们可以维护一个有兴趣了解司机位置的客户(订阅者)列表,并且每当我们在 DriverLocationHT 中
有该司机的更新时,我们就可以向所有订阅的客户广播司机的当前位置。这样,我们的系统可确保始终向客户显示驾驶员的
当前位置。

我们需要多少内存来存储所有这些订阅?正如我们上面估计的,我们将拥有 100 万日活跃客户和 50 万日活跃司机。平均而
言,我们假设有 5 名客户订阅了一名司机。假设我们将所有这些信息存储在哈希表中,以便我们

可以高效更新。我们需要存储司机和客户 ID 来维护订阅。

假设我们需要 3 个字节用于 DriverID,8 个字节用于 CustomerID,那么我们将需要 21MB
记忆。

$$(500K * 3) + (500K * 5 * 8) \approx 21 \text{ MB}$$

我们需要多少带宽才能向客户广播司机的位置?对于每个活跃的司机,我们有五个订阅者,因此我们拥有的订阅者总数:

$$5 * 500K \Rightarrow 2.5M$$

我们需要每秒向所有这些客户发送 DriverID (3 字节)及其位置 (16 字节),因此,我们需要以下带宽:

2.5M * 19 字节 => 47.5 MB/s

如何高效地实现Notification服务?我们可以使用 HTTP 长轮询或推送通知。

如何为当前客户添加新的发布商/驱动程序?正如我们上面提出的,客户第一次打开 Uber 应用程序时会订阅附近的司机,当新司机进入客户正在查看的区域时会发生什么?要动态添加新的客户/司机订阅,我们需要跟踪客户正在观看的区域。

这将使我们的解决方案变得复杂;如果客户端不推送这些信息,而是从服务器拉取它,怎么样?

如果客户端从服务器获取附近驱动程序的信息怎么样?客户端可以发送自己的当前位置,服务器将从四叉树中找到附近的所有驱动程序并将其返回给客户端。收到此信息后,客户可以更新屏幕以反映驾驶员的当前位置。客户端可以每五秒查询一次,以限制到服务器的往返次数。与上述的推送模型相比,该解决方案看起来更简单。

当网格达到最大限制时,我们是否需要立即重新分区?在我们决定划分它之前,我们可以有一个缓冲,让每个网格增长到超出极限一点。假设我们的网格在分区/合并之前可以额外增长/缩小 10%。这应该会减少网格分区的负载或合并高流量网格。

“叫车”用例如何运作?

1. 顾客提出乘车请求。
2. 其中一台聚合器服务器将接受请求并要求附近的四叉树服务器返回司机。

3. 聚合器服务器收集所有结果并按评级对它们进行排序。
4. 聚合器服务器将同时向排名靠前（例如三个）的司机发送通知，首先接受请求的司机将被分配行程。其他司机将收到取消请求。如果三个驱动程序均未响应，聚合器将请求

从列表中接下来的三位司机中选择骑行。

5. 一旦司机接受请求，客户就会收到通知。

5. 容错和复制

如果驾驶员位置服务器或通知服务器挂掉了怎么办？我们需要这些服务器的副本，以便在主服务器死亡时，辅助服务器可以接管控制权。此外，我们可以将这些数据存储在一些持久存储中，例如可以提供快速 IO 的 SSD；这将确保如果主服务器和辅助服务器都死掉，我们可以从持久存储中恢复数据。

六、排名

如果我们不仅想根据邻近度还想根据流行度或相关性对搜索结果进行排名，该怎么办？

我们如何才能返回给定半径内评分最高的司机？假设我们在数据库和四叉树中跟踪每个驱动程序的总体评分。汇总的数字可以代表我们系统中的受欢迎程度，例如，一个司机在十颗星中获得多少颗星？在搜索给定半径内的前 10 个驱动程序时，我们可以要求四叉树的每个分区返回具有最大评级的前 10 个驱动程序。然后，聚合器服务器可以确定不同分区返回的所有驱动程序中的前 10 个驱动程序。

7. 高级问题

1. 我们将如何处理网络缓慢且断线的客户端？
2. 如果客户在乘车时断线怎么办？我们将如何处理计费这样的场景？
3. 与服务器总是推送信息相比，客户端提取所有信息怎么样？

设计 Ticketmaster (*新*)

让我们设计一个在线售票系统来销售电影票，例如 Ticketmaster 或 BookMyShow。

类似服务：bookmyshow.com、ticketmaster.com 难度级别：困难

1. 什么是在线电影票预订系统?

电影票预订系统使客户能够在线购买剧院座位。电子票务系统允许客户随时随地浏览当前正在播放的电影并预订座位。

2. 系统的要求和目标

我们的机票预订服务应满足以下要求：

功能要求：

1. 我们的订票服务应该能够列出其加盟影院所在的不同城市位于。
2. 一旦用户选择了城市，服务应该显示在该特定城市发布的电影城市。
3. 一旦用户选择了一部电影，该服务应该显示正在播放该电影的电影院及其可用的演出时间。
4. 用户应该能够选择特定电影院的演出并预订门票。
5. 该服务应该能够向用户显示电影院大厅的座位安排。这用户应该能够根据自己的喜好选择多个座位。
6. 用户应该能够区分可用座位和预订座位。
7. 用户应该能够在付款完成预订之前保留座位五分钟。

8. 如果有可能有空位，用户应该能够等待，例如：当其他用户持有的到期时。
9. 应以公平、先到先服务的方式为等候的顾客提供服务。

非功能性要求：

1. 系统需要高并发。将会有多个预订请求在任何特定时间点的同一个座位。该服务应该优雅、公平地处理这个问题。
2. 服务的核心是机票预订，也就是金融交易。这意味着系统应该是安全的并且数据库应该符合 ACID。

3. 一些设计考虑

1. 为简单起见，我们假设我们的服务不需要任何用户身份验证。
2. 系统不处理部分机票订单。用户要么获得他们想要的所有门票，要么他们什么也得不到。
3. 系统必须公平。
4. 为了防止系统滥用，我们可以限制用户一次预订十个以上的座位。
5. 我们可以假设热门/期待已久的电影上映和座位上的流量会激增会很快填满。该系统应该具有可扩展性和高可用性，以跟上流量激增。

4. 容量估算

流量估算:假设我们的服务每月有 30 亿次页面浏览量,每月售出 1000 万张门票。

存储估算：假设我们有 500 个城市，平均每个城市有 10 家电影院。如果每个电影院有 2000 个座位，平均每天有两场演出。

假设每个座位预订需要 50 个字节 (ID、NumberOfSeats、ShowID、MovieID、SeatNumbers、SeatStatus、Timestamp 等) 来存储在数据库中。我们还需要存储有关电影和电影院的信息;我们假设它需要 50 个字节。因此,要存储有关所有内容的所有数据

一天内所有城市所有电影院的放映时间：

500个城市 * 10个电影院 * 2000个座位 * 2场演出 * (50+50)字节 = 2GB/天

要存储五年的数据,我们需要大约 3.6TB。

5. 系统API

我们可以使用 SOAP 或 REST API 来公开我们服务的功能。以下是搜索电影节目和预订座位的 API 的定义。

SearchMovies(api_dev_key、关键字、城市、经纬度、半径、开始日期时间、结束日期时间、邮政编码、包括拼写检查、每页结果、排序顺序)

参数：

api dev key (string):注册账户的API开发者密钥。除其他外,这将用于根据分配的配额限制用户。

关键字 (字符串): 要搜索的关键字。

`city` (字符串) : 过滤电影所依据的城市。

lat_long (字符串) :要过滤的纬度和经度。 **radius** (数字) :我们所在区域的半径
想要搜索事件。

`start_datetime` (字符串) · 使用开始日期时间过滤电影。

`end_datetime` (字符串) : 过滤具有结束日期时间的电影

postal_code (字符串) : 按邮政编码/邮政编码过滤由影

`includeSpellcheck` (枚举：“yes”或“no”)：是否在响应中包含拼写检查建议

results_per_page (number): 每页返回的结果数，最大值为 20。

`sorting_order` (string) 搜索结果的排序顺序。一些允许的值： `名称,asc` , `名称,desc` , `日期,asc` , `日期,desc` , `距离,asc` , `名称,日期,asc` , `名称,日期,desc` , `日期,名称,asc` , `日期,名称,desc` .

返回：(JSON)

以下是电影及其节目的示例列表：

```
[  
 {  
     “电影ID” :1,  
     “显示ID” :1,  
     “标题”：“汽车总动员2” ,
```

“描述”：“关于汽车”、“持续时间”：120、“类型”：“动画”、“语言”：“英语”、“发布日期”：“2014年10月8日”、“国家”：美国、“开始时间”：14:00，结束时间：16:00，座位：[

```
{
    “类型”：“常规”
    “价格”：14.99 “状态：
    “几乎满了”
}, {
```

```
    “类型”：“高级”
    “价格”：24.99 “状态：
    “可用”
}
```

```
], {
    “MovieID”：1,
    “ShowID”：2，“标题”：“汽车总动员 2”，“描述”：“关于汽车”，“持续时间”：120，“类型”：“动画”，“语言”：“英语”，“发布日期”：“2014年10月8日”，“国家”：美国，“开始时间”：16:30，结束时间：18:30，座位：[
```

```
{
    “类型”：“常规”
    “价格”：14.99 “状态：
    “已满”
}, {
```

```
    “类型”：“高级”
    “价格”：24.99 “状态：
    “几乎满了”
}
```

```
], ]
ReserveSeats(api_dev_key、session_id、movie_id、show_id、seat_to_reserve[])
参数：
```

api_dev_key（字符串）：与上面相同

session_id（字符串）：用于跟踪此预订的用户会话 ID。一旦预约时间到期，用户在服务器上的预约将使用该ID被删除。 movie_id（字符串）：要预订的电影。

show_id (字符串) : 显示要预订的内容。 Seats_to_reserve (number): 包含要预订的座位 ID 的数组。

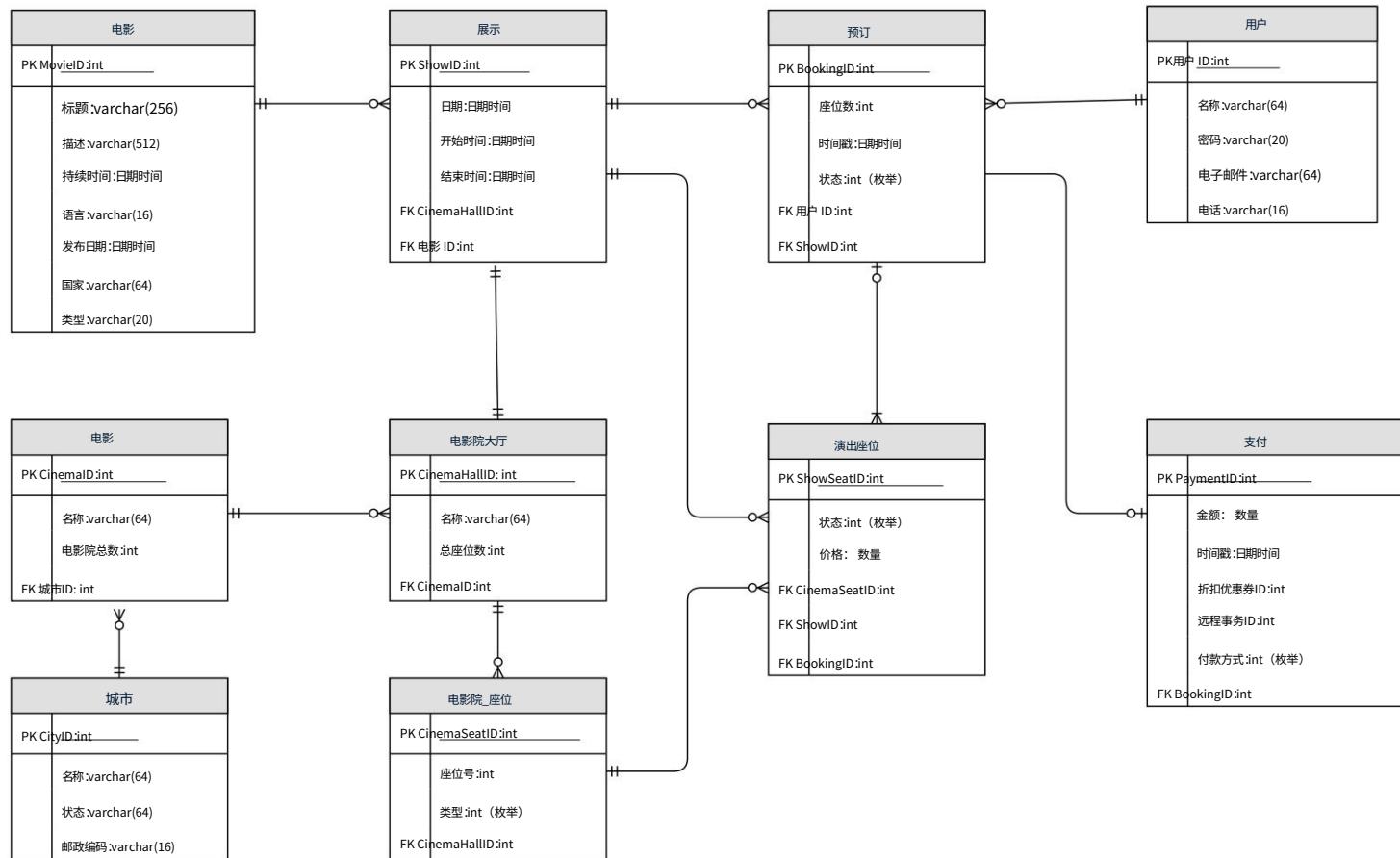
返回: (JSON)

返回预订的状态,可能是以下之一: 1) “预订成功”
2) “预订失败 - 显示已满”,3) “预订失败 - 重试,因为其他用户正在保留预订的座位”。

6. 数据库设计以下是关于

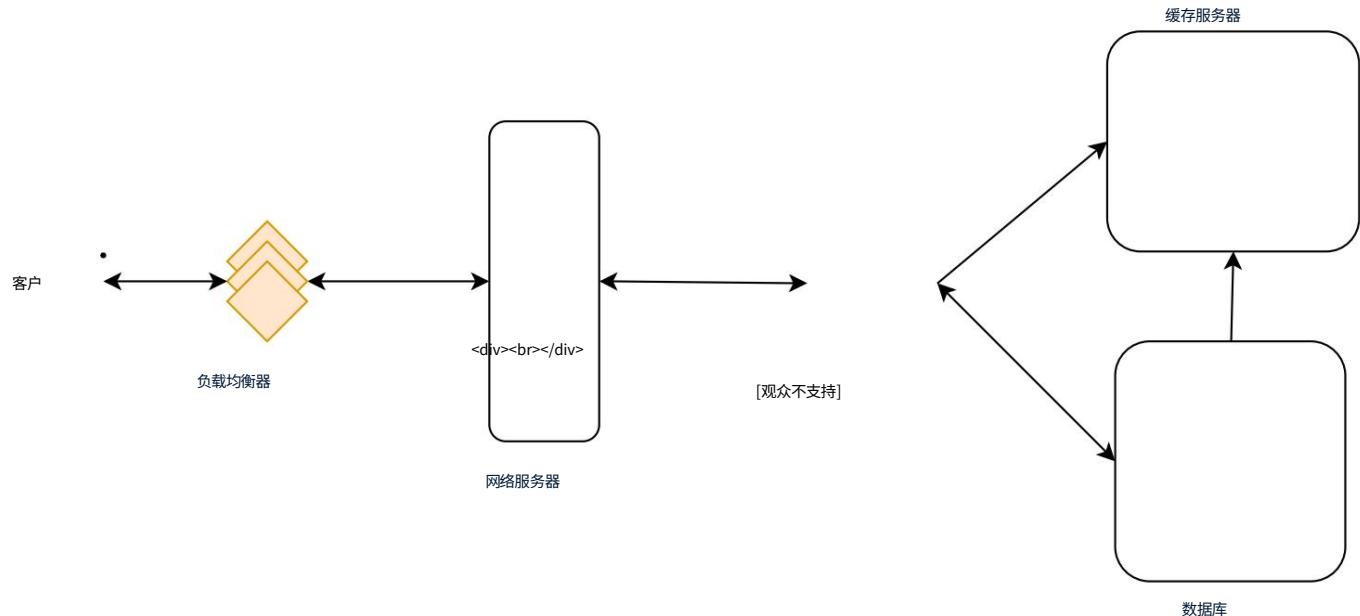
我们要存储的数据的一些观察:

1. 每个城市可以有多个电影院。
2. 每个电影院都会有很多放映厅。
3. 每部电影都会有很多场演出,每个演出都会有很多预订。
4. 一个用户可以有很多预订。



7. 高层设计

在高层,我们的网络服务器将管理用户的会话,应用程序服务器将处理所有票务管理、将数据存储在数据库中以及与缓存服务器一起处理预订。



8. 详细组件设计

首先,让我们尝试构建我们的服务,假设它是由单个服务器提供服务的。

机票预订工作流程:以下是典型的机票预订工作流程:

1. 用户搜索电影。
2. 用户选择电影。
3. 向用户显示电影的可用节目。
4. 用户选择节目。
5. 用户选择要预订的座位数。
6. 如果有所需的座位数,则会向用户显示剧院地图以供选择座位。如果没有,用户将被带到下面的“步骤 8”。
7. 一旦用户选择了座位,系统将尝试保留那些选定的座位。
8. 如果无法预订座位,我们有以下选择:
 - 演出已满;用户会看到错误消息。 · 用户想要预订的座位已不再可用,但还有其他座位可用,因此用户被带回到剧院地图以选择不同的座位。
 - 没有座位可供预订,但所有座位尚未预订,因为还有一些座位其他用户在预订池中持有且尚未预订的座位。用户将

被带到等待页面,他们可以在那里等待,直到所需的座位从预订池中释放出来。这种等待可能会导致以下选择:

- 如果有所需数量的座位,用户将被带到剧院地图
他们可以选择座位的页面。
- 等待时,如果所有座位都已被预订或预订池中的座位少于
用户打算预订,则会向用户显示错误消息。 · 用户取消等待并返回电影搜索页
面。 · 用户最多可以等待一小时,在该用户的会话过期并且用户
返回到电影搜索页面。

9. 如果座位预订成功,用户有五分钟的时间支付预订费用。后

付款后,预订即显示完成。如果用户无法在五分钟内付款,则其所有
保留的座位可以释放给其他用户使用。

1共 8 个

服务器如何跟踪所有尚未预订的活动预订?服务器如何跟踪所有等待的顾客?

我们需要两个守护程序服务,一个用于跟踪所有活动预订并从系统中删除任何过期的预订;另一个用于跟踪所有活动预订并从系统中删除所有过期预订。我们称之为ActiveReservationService。另一项服务将跟踪所有等待用户的请求,一旦所需数量的座位可用,它将通知(等待时间最长的)用户选择座位;我们称之为WaitingUserService。

A. 主动预订服务

我们可以将内存中“演出”的所有保留保留在类似于[Linked HashMap](#) 的数据结构中
或[树形图](#)除了将所有数据保存在数据库中之外。我们需要一个链接的 HashMap 类型的数据结构,它允许我们跳转到任何预订,以便在预订完成时将其删除。
此外,由于我们将与每个预订相关联的到期时间,因此 HashMap 的头部将始终指向最旧的预订记录,以便在达到超时时预订可以过期。

为了存储每个节目的每个预订,我们可以有一个哈希表,其中“键”是“ShowID”,“值”是包含“BookingID”和创建“时间戳”的连接哈希映射。

在数据库中,我们将把预订存储在“预订”表中,到期时间将在时间戳列中。“Status”字段的值为“Reserved (1)”,一旦预订完成,系统会将“Status”更新为“Booked (2)”,并从 Linked HashMap 中删除预订记录的相关节目。当预订过期时,除了从内存中删除它之外,我们还可以将其从 Booking 表中删除,或者将其标记为“Expired (3)”。

ActiveReservationsService 还将与外部金融服务合作处理用户付款。

每当预订完成或预订过期时,WaitingUsersService 都会收到信号,以便可以为任何等待的客户提供服务。

核心价值

[观众不支持]LinkedHashMap<预订ID、时间戳>
例如,
[(1,1499818500), (2,1499818800), (3,1499818800)]< 123

ActiveReservationsService 跟踪所有活动预订

b.等待用户服务

就像 ActiveReservationsService 一样,我们可以将演出的所有等待用户保存在内存中的 LinkedHashMap 或 TreeMap 中。我们需要一个类似于 LinkedHashMap 的数据结构,以便当用户取消请求时,我们可以跳转到任何用户,将其从 HashMap 中删除。另外,由于我们以先到先服务的方式服务,LinkedHashMap 的头部总是指向等待时间最长的用户,这样每当有座位时,我们就可以公平地为用户提供服务。

我们将有一个哈希表来存储每场演出的所有等待用户。“键”将是“ShowID”,“值”将是包含“UserID”及其等待开始时间的链接 HashMap。

客户端可以使用[长轮询](#)以便及时了解预订状态。每当座位可用时,服务器就可以使用此请求来通知用户。

预订有效期

在服务器上,ActiveReservationsService 跟踪活动预订的到期时间(基于预订时间)。由于客户端将显示一个计时器(用于过期时间),这可能与服务器有点不同步,因此我们可以在服务器上添加五秒的缓冲区,以防止体验中断,这样客户端永远不会在服务器超时后,无法成功购买。

9.并发性

如何处理并发性,以便没有两个用户能够预订相同的座位。我们可以在 SQL 数据库中使用事务来避免任何冲突。例如,如果我们使用 SQL 服务器,我们可以利用事务隔离级别在更新行之前锁定它们。这是示例代码:

设置事务隔离级别可串行化;

开始交易;

```
-- 假设我们打算为 ShowID=99 保留三个座位 (ID:54,55,56)
选择 *      来自 Show_Seat, 其中 ShowID=99 && ShowSeatID in (54, 55, 56) &&
状态=0  免费
```

-- 如果上述语句返回的行数为三,我们可以更新为 -- 返回成功,否则返回失败给用户。

更新 Show_Seat ...

更新预订...

提交交易;

“可串行化”是最高级别的隔离级别,可保证安全,免受[Dirty](#)、[不可重复](#),和[幻影读](#)。这里需要注意一件事;在事务中,如果我们读取行,我们就会获得对它们的写锁定,以便其他任何人都无法更新它们。

一旦上述数据库事务成功,我们就可以开始跟踪预订
主动预订服务。

10. 容错

当 ActiveReservationsService 或 WaitingUserService 崩溃时会发生什么?

每当 ActiveReservationsService 崩溃时,我们都可以从 “Booking”表中读取所有活动预订。请记住,在预订完成之前,我们会将“状态”列保留为“已预订 (1)”。另一种选择是采用主从配置,这样当主设备崩溃时,从设备可以接管。我们没有将等待用户存储在数据库中,因此,当 WaitingUserService

崩溃时,除非我们有主从设置,否则我们没有任何方法来恢复该数据。

同样,我们将为数据库设置主从设置,以使其具有容错能力。

11. 数据分区

数据库分区:如果我们按 “MovieID”分区,那么一部电影的所有节目都将位于一个服务器。对于一部非常热门的电影,这可能会导致该服务器产生大量负载。更好的方法是根据ShowID进行分区;这样,负载就分布在不同的服务器上。

ActiveReservationService 和 WaitingUserService 分区:我们的 Web 服务器将管理所有活动用户的会话并处理与用户的所有通信。我们可以使用一致

根据 “ShowID”为 ActiveReservationService 和 WaitingUserService 分配应用程序服务器的哈希值。这样,特定节目的所有预订和等待用户都将由一组特定的服务器处理。让我们假设为了负载平衡,我们的一致性哈希为任何节目分配三个服务器,因此每当预订过期时,持有该预订的服务器将执行以下操作:

1. 更新数据库以删除预订 (或将其标记为过期)并更新 “Show_Seats”表中的座位状态。
2. 从 LinkedHashMap 中删除保留。
3. 通知用户其预订已过期。

4. 向所有正在等待该节目的用户的 WaitingUserService 服务器广播一条消息,以找出等待时间最长的用户。一致的哈希方案将告诉哪些服务器正在保存这些用户。

5. 向持有等待时间最长的用户的WaitingUserService服务器发送消息进行处理
他们请求所需的座位是否有空位。

预订成功后,将会发生以下情况:

1. 持有该预订的服务器向所有持有该节目的等待用户的服务器发送消息,以便这些服务器可以终止所有需要比可用座位更多座位的等待用户。
2. 收到上述消息后,所有持有等待用户的服务器都会查询数据库,了解现在有多少个空闲席位。数据库缓存将极大地帮助您仅运行此查询一次。
3. 使所有想要预订的座位多于可用座位的等待用户过期。为此,WaitingUserService 必须迭代所有等待用户的 LinkedHashMap。

其他资源

以下是一些可供进一步阅读的有用链接:

1. [发电机](#)- 高可用的键值存储
2. [卡夫卡](#)- 用于日志处理的分布式消息系统
3. [一致性哈希](#)- 原纸
4. [帕克索斯](#)- 分布式共识协议
5. [并发控制](#)- 并发控制的乐观方法
6. [八卦协议](#)- 用于故障检测等。
7. [胖乎乎的](#)- 松耦合分布式系统的锁服务
8. [动物园管理员](#)- 互联网规模系统的无等待协调
9. [映射减少](#)- 简化大型集群上的数据处理
10. [Hadoop](#)- 分布式文件系统

系统设计基础知识

每当我们设计一个大型系统时,我们需要考虑以下几点:

1. 可以使用哪些不同的建筑部件?
2. 这些部分如何相互配合?
3. 我们如何才能最好地利用这些部分:正确的权衡是什么?

在需要扩展之前进行投资通常不是一个明智的商业主张;然而,对设计的一些深思熟虑可以在未来节省宝贵的时间和资源。在接下来的章节中,我们将尝试定义可扩展系统的一些核心构建块。熟悉这些

概念对于理解分布式系统概念将非常有帮助。在下一节中,我们将介绍一致性哈希、CAP 定理、负载平衡、缓存、数据分区、索引、代理、队列、复制以及在 SQL 与 NoSQL 之间进行选择。

让我们从分布式系统的关键词开始。

分布式的主要特征

系统

分布式系统的关键特征包括可扩展性、可靠性、

可用性、效率和可管理性。让我们简单回顾一下它们:

可扩展性

可扩展性是系统、流程或网络不断发展和扩展的能力

管理增加的需求。任何可以连续地进行的分布式系统

为了支持不断增长的工作量而发展被认为是

可扩展。

由于多种原因(例如数据增加),系统可能必须进行扩展

工作量或增加的工作量,例如交易数量。可扩展的

系统希望在不损失性能的情况下实现这种扩展。

一般来说,系统的性能,尽管设计(或声称)是为了

具有可扩展性,由于管理或

环境成本。例如,网络速度可能会变慢,因为机器往往彼此相距较远。更一般地说,一些任务可能不是分布式的,要么因为它们固有的原子性质,要么因为系统设计上的一些缺陷。在某些时候,此类任务将限制通过分发获得的加速。可扩展的架构避免了这种情况并尝试平衡所有参与节点上的负载均匀。

水平缩放与垂直缩放:水平缩放意味着您可以缩放在资源池中添加更多服务器,同时垂直扩展意味着您可以通过向设备添加更多功能(CPU、RAM、存储等)来进行扩展现有服务器。

通过水平缩放,通过添加更多内容通常更容易动态缩放机器放入现有池中;垂直缩放通常仅限于单个服务器的容量以及超出该容量的扩展通常涉及停机时间有上限。

[Cassandra](#)是水平缩放的好例子和[MongoDB](#)像他们两者都提供了一种通过添加更多机器来水平扩展的简单方法满足不断增长的需求。同样,垂直扩展的一个很好的例子是 MySQL因为它允许通过从小到大切换来实现垂直扩展的简单方法更大的机器。然而,这个过程通常会涉及停机。

垂直缩放

向同一服务器添加更多资源

水平缩放

添加更多服务器

与

垂直缩放与水平缩放

可靠性

根据定义,可靠性是指系统在给定时间内发生故障的概率。

简单来说,如果分布式系统保持

即使其一个或多个软件或硬件也提供其服务

组件发生故障。可靠性是任何设备的主要特征之一

分布式系统,因为在这样的系统中,任何发生故障的机器总是可以

替换为另一名健康的人,确保完成要求的任务

任务。

以大型电子商务商店 (如[亚马逊](#))为例,在哪里

主要要求之一是任何用户事务都不应该被

由于运行该事务的机器出现故障而被取消。为了

例如,如果用户将商品添加到购物车中,系统将预计不会失去它。可靠的分布式系统通过以下方式实现这一目标软件组件和数据的冗余。如果服务器携带用户的购物车出现故障,另一台具有该购物车精确副本的服务器购物车应该取代它。显然,冗余是有成本的,可靠的系统必须付出代价通过消除每一点来实现服务的弹性失败。

可用性

根据定义,可用性是系统保持运行以执行任务的时间其在特定时期所需要的功能。这是一个简单的衡量系统、服务或机器保持运行的时间百分比正常情况下。可以飞行多个小时的飞机一个月没有太多停机时间,可以说具有很高的可用性。可用性考虑了可维护性、维修时间、备件可用性、以及其他物流考虑因素。如果一架飞机因维修而停机,在此期间被认为不可用。可靠性是指随着时间的推移,考虑到所有可能的真实情况的可用性。可能发生的任何状况。一架可以穿越任何地方的飞机可能的安全天气比有漏洞的天气更可靠可能的条件。可靠性对比可用性如果一个系统是可靠的,那么它就是可用的。不过,如果说的话,也不是必然可靠。换句话说,高可靠性有助于高可用性,但即使使用通过最大限度地减少维修时间并确保备件到位来消除不可靠的产品

在需要时随时可用。我们以网上的例子为例
零售店开业后头两年的可用性高达 99.99%。
然而,该系统在没有进行任何信息安全测试的情况下就启动了。
客户对系统感到满意,但他们没有意识到事实并非如此
非常可靠,因为它容易受到可能的风险的影响。第三年,系统
遭遇一系列信息安全事件,突然导致
长时间内可用性极低。这导致
对客户造成声誉和经济损失。

效率

要了解如何衡量分布式系统的效率,让我们
假设我们有一个以分布式方式运行并交付的操作
结果是一组项目。其效率的两个标准衡量标准是
响应时间 (或延迟) ,表示获取第一项的延迟,以及
吞吐量 (或带宽) ,表示交付的物品数量
在给定的时间单位 (例如,秒)内。这两项措施分别对应
以下单位成本:

- 系统节点全局发送的消息数
无论消息大小如何。
- 表示数据交换量的消息大小。

分布式数据结构支持的操作的复杂性 (例如,
在分布式索引中搜索特定键)可以表征为
这些成本单位之一的函数。一般来说,分析一个
就 “消息数量”而言的分布式结构过于简单化。它
忽略了很多方面的影响,包括网络拓扑、
网络负载及其变化,软件可能的异构性

以及涉及数据处理和路由等的硬件组件。

然而,开发一个精确的成本模型是相当困难的。

准确地考虑所有这些性能因素;因此,我们

必须接受对系统行为的粗略但稳健的估计。

可维护性或可管理性

设计分布式系统时另一个重要的考虑因素是如何

操作和维护方便。可服务性或可管理性是

修复或维护系统的简单性和速度;如果

修复故障系统的时间增加,可用性就会降低。

可管理性需要考虑的因素是诊断的难易程度和

了解问题发生时的情况、易于进行更新或

修改,以及系统操作的简单程度(即,是否定期进行

运行无故障或异常?)。

及早发现故障可以减少或避免系统停机。为了

例如,一些企业系统可以自动呼叫服务中心

(无需人工干预)当系统遇到系统故障时。

负载均衡

负载均衡器(LB)是任何分布式系统的另一个关键组件。它

有助于将流量分散到服务器集群上,以改善

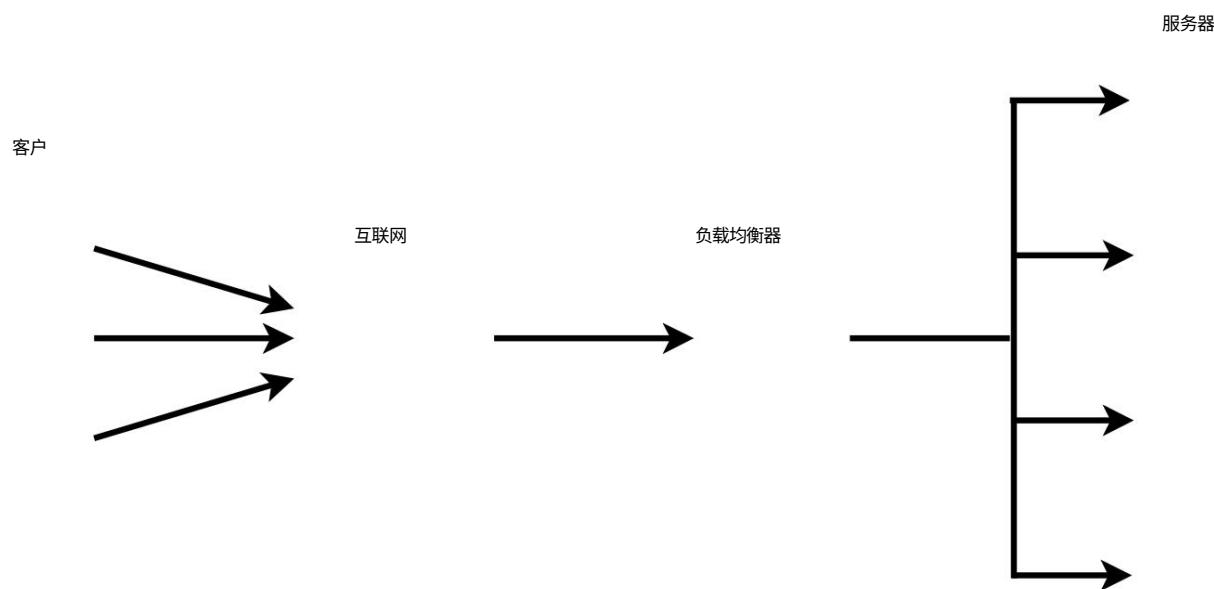
应用程序、网站或数据库的响应能力和可用性。磅

在分发请求时还跟踪所有资源的状态。

如果服务器无法接受新请求或没有响应或已

错误率升高,LB将停止向此类服务器发送流量。

通常,负载均衡器位于客户端和服务器之间,接受传入网络和应用程序流量并将流量分配给使用各种算法的多个后端服务器。通过平衡应用跨多个服务器的请求,负载均衡器减少了单个服务器的数量负载并防止任何一台应用程序服务器成为单点故障,从而提高整体应用程序可用性和响应能力。



为了充分利用可扩展性和冗余性,我们可以尝试平衡负载

系统的每一层。我们可以在三个地方添加 LB:

- 用户和网络服务器之间
- Web 服务器和内部平台层 (例如应用程序)之间
- 服务器或缓存服务器
- 内部平台层和数据库之间。

负载平衡的好处

- 用户体验更快、不间断的服务。用户不必等待单个陷入困境的服务器完成其之前的任务。反而，他们的请求会立即传递给更容易获得的资源。
- 服务提供商体验到更少的停机时间和更高的吞吐量。即使服务器完全故障也不会影响最终用户体验，因为负载均衡器将简单地将其路由到健康的服务器。
- 负载平衡使系统管理员更容易处理传入请求，同时减少用户的等待时间。
- 智能负载平衡器提供预测分析等优势在交通瓶颈发生之前确定它们。结果，聪明的负载均衡器为组织提供了可操作的见解。这些是关键自动化并有助于推动业务决策。
- 系统管理员遇到的故障或压力组件更少。负载平衡不是由单个设备执行大量工作，而是多个设备执行一些工作。

负载均衡算法

负载均衡器如何选择后端服务器?

负载均衡器在将请求转发到负载均衡器之前会考虑两个因素

后端服务器。他们首先会确保他们选择的服务器实际上是

适当地响应请求,然后使用预先配置的

从一组健康服务器中选择一个的算法。我们将讨论这些

算法很快。

健康检查- 负载均衡器应该只将流量转发到“健康”

后端服务器。为了监控后端服务器的健康状况,“健康检查”

定期尝试连接后端服务器以确保服务器正常

听。如果服务器未通过健康检查,则会自动从服务器中删除

池,并且在其健康状况响应之前,流量不会转发给它

再次检查。

负载均衡的方法有很多种,使用的算法也不同

针对不同的需求。

·最少连接方法 此方法将流量定向到服务器

具有最少的活动连接。这种方法在以下情况下非常有用

有大量的持久客户端连接

服务器之间分布不均匀。

·最小响应时间方法 该算法将流量引导至

活动连接数最少且平均数最低的服务器

响应时间。

·最小带宽方法- 该方法选择最小带宽方法

目前提供的流量最少 (以兆位为单位)

第二 (Mbps) 。

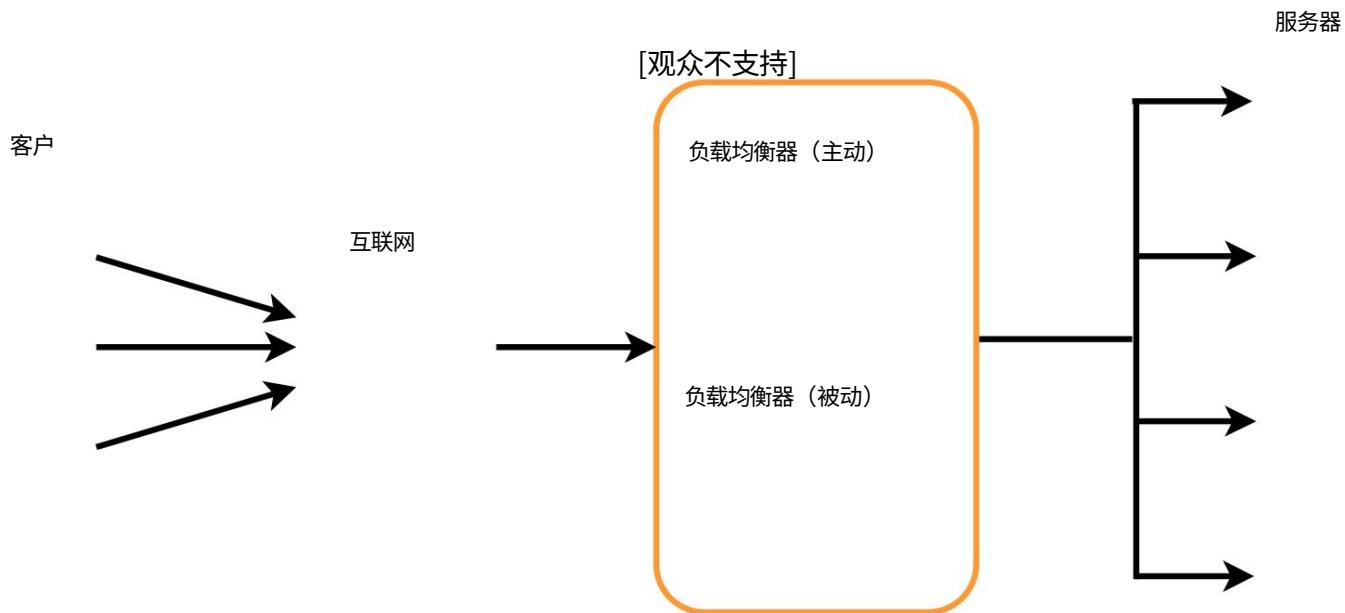
· 循环方法 此方法循环遍历服务器列表并
将每个新请求发送到下一个服务器。当它到达终点时
列表,它从头开始。当
服务器规格均等,持久性不多
连接。

· 加权循环法 加权循环调度
旨在更好地处理不同处理的服务器
能力。每个服务器都分配有一个权重（一个整数值，
表示处理能力）。权重较高的服务器接收
新连接先于权重较小的连接和权重较高的服务器
权重比权重较小的连接获得更多连接。

· IP 哈希值 在此方法下,客户端 IP 地址的哈希值是
计算将请求重定向到服务器。

冗余负载均衡器

负载均衡器可能会出现单点故障;为了克服这个问题,第二个
负载均衡器可以连接到第一个以形成集群。每个LB
监控对方的健康状况,因为他们两人能力相同
服务流量和故障检测,如果主负载均衡器
失败,第二个负载均衡器接管。



以下链接有一些关于负载均衡器的很好的讨论：

[1][什么是负载均衡](#)

[2][系统架构简介](#)

[3][负载均衡](#)

缓存

负载平衡可帮助您在不断增加的规模中水平扩展

服务器数量,但缓存将使您能够更好地利用

您已经拥有的资源以及其他方式无法获得的资源

产品要求可行。缓存利用了局部性

参考原则:最近请求的数据很可能被再次请求。

它们几乎用于计算的每一层:硬件、操作

系统、Web 浏览器、Web 应用程序等。缓存很短

术语记忆:它的空间有限,但速度通常比

原始数据源并包含最近访问的项目。

缓存可以存在于体系结构的各个级别,但通常出现在
距离前端最近,实现快速返回数据
而不对下游层面征税。

应用服务器缓存

将缓存直接放置在请求层节点上,可以实现本地存储
响应数据。每次向服务发出请求时,节点都会
快速返回本地缓存数据 (如果存在)。如果缓存中没有,则
请求节点将从磁盘查询数据。一次请求的缓存
层节点也可以位于内存中 (速度非常快) 和
节点的本地磁盘 (比访问网络存储更快)。
当你将其扩展到许多节点时会发生什么?如果请求层是
扩展到多个节点,每个节点都有主机还是很有可能的
它自己的缓存。但是,如果您的负载均衡器随机分配请求
跨节点,同一个请求会到不同的节点,从而增加
缓存未命中。克服这个障碍的两种选择是全局缓存和
分布式缓存。

内容分发网络 (CDN)

CDN 是一种缓存,适用于提供大量服务的网站
静态媒体。在典型的 CDN 设置中,请求将首先向 CDN 询问
一块静态媒体;如果本地有该内容,CDN 将提供该内容
可用的。如果不可用,CDN 将向后端服务器查询
文件,将其缓存在本地,然后将其提供给请求的用户。
如果我们正在构建的系统还不够大,无法拥有自己的 CDN,我们
可以通过单独提供静态媒体来简化未来的过渡

子域（例如static.yourservice.com）使用轻量级 HTTP 服务器，例如 Nginx，稍后将 DNS 从您的服务器切换到 CDN。

缓存失效

虽然缓存非常棒，但它确实需要一些维护来保持缓存与事实来源（例如数据库）一致。如果数据是在数据库中修改，应该在缓存中失效；如果没有，这可能会导致应用程序行为不一致。

解决这个问题称为缓存失效；主要有以下三个使用的方案：

Write-through 缓存：在这种方案下，数据被写入缓存并同时对应的数据库。缓存的数据允许快速检索，并且由于相同的数据被写入永久存储，我们将在缓存和存储之间实现完全的数据一致性贮存。此外，该方案还确保在发生意外时不会丢失任何东西崩溃、电源故障或其他系统中断。

尽管如此，直写可以最大限度地降低数据丢失的风险，因为每次写入操作必须进行两次才能将成功返回给客户端，这该方案的缺点是写操作延迟较高。

Write-around 缓存：这种技术类似于 write through 缓存，但是数据绕过缓存直接写入永久存储。这个可以减少写入操作淹没的缓存

随后会被重新读取，但缺点是读取请求最近写入的数据将创建“缓存未命中”并且必须从中读取后端存储速度较慢并且延迟较高。

回写式缓存：在这种方案下，数据单独写入缓存，完成后立即向客户确认。写入到

永久存储是在指定的时间间隔后或在一定的条件下完成的状况。这将为写入密集型应用带来低延迟和高吞吐量然而,这种速度会带来数据丢失的风险崩溃或其他不良事件,因为写入数据的唯一副本位于缓存。

缓存驱逐策略

以下是一些最常见的缓存逐出策略：

1.先进先出（FIFO）:缓存逐出最先访问的第一个块

不考虑访问频率或次数

前。

2. 后进先出（LIFO）:缓存逐出访问次数最多的块

最近第一次,不管有多少次

之前访问过。

3. 最近最少使用（LRU）:首先丢弃最近最少使用的项目。

4. 最近使用的（MRU）:丢弃,与LRU相比,最常使用的

首先是最近使用过的物品。

5. 最不频繁使用 (LFU):计算需要某项的频率。

最不常用的会首先被丢弃。

6.随机替换 (RR):随机选择一个候选项并

必要时丢弃它以腾出空间。

以下链接有一些关于缓存的很好的讨论：

[1][缓存](#)

[2][系统架构简介](#)

分片或数据分区

数据分区（也称为分片）是一种分解大数据的技术。

数据库（DB）分成许多更小的部分。这是一个分裂的过程

DB/表跨多台机器提高可管理性，

应用程序的性能、可用性和负载平衡。这

数据分片的理由是，在一定规模之后，它会更便宜

通过添加更多机器来水平扩展比

通过添加更强大的服务器来垂直扩展它。

1. 划分方法

人们可以使用多种不同的方案来决定如何分手

将一个应用程序数据库分解为多个较小的数据库。下面是其中的三个

各种大型应用程序使用的最流行的方案。

A。水平分区：在该方案中，我们将不同的行放入不同的行中

表。例如，如果我们在表中存储不同的位置，我们可以

决定将邮政编码小于 10000 的位置存储在一张表中

邮政编码大于 10000 的地点存储在单独的表中。

这也称为基于范围的分片，因为我们存储不同的范围

不同表中的数据。

这种方法的关键问题是，如果使用范围的值

如果没有仔细选择分片，那么分区方案将导致

服务器不平衡。在前面的示例中，根据位置分割

邮政编码假设地点将均匀分布在不同的地区

邮政编码。这个假设是不成立的，因为a中会有很多地方

与郊区城市相比，曼哈顿等人口稠密的地区。

b. 垂直分区 : 在这个方案中 , 我们将数据划分为存储表

与他们自己的服务器中的特定功能相关。例如 , 如果我们是

构建类似 Instagram 的应用程序 - 我们需要在其中存储相关数据

用户、他们上传的照片以及他们关注的人 - 我们可以决定放置

一台数据库服务器上的用户个人资料信息 , 另一台数据库服务器上的好友列表以及照片

在第三台服务器上。

垂直分区实现简单 , 对系统影响小

应用程序。这种方法的主要问题是 , 如果我们的应用程序

经历额外的增长 , 那么可能有必要进一步划分

具有跨不同服务器的特定数据库 (例如 , 不可能

单个服务器可处理 140 倍 100 亿张照片的所有元数据查询

万用户) 。

C. 基于目录的分区 : 一种松散耦合的解决方法

上述方案中提到的问题是创建一个查找服务

了解您当前的分区方案并将其从数据库中抽象出来

访问代码。因此 , 为了找出特定数据实体所在的位置 , 我们进行查询

保存每个元组键与其数据库之间的映射的目录服务器

服务器。这种松散耦合的方法意味着我们可以执行类似的任务

将服务器添加到数据库池或更改我们的分区方案 , 而无需

对申请有影响。

2. 划分标准

A. 基于键或哈希的分区 : 在此方案下 , 我们应用哈希

对我们存储的实体的一些关键属性起作用 ; 产生

分区号。例如 , 如果我们有 100 个数据库服务器 , 并且我们的 ID 是

每次有新记录时该数值就会增加 1

插入。在此示例中 , 哈希函数可以是 “ $ID \% 100$ ” , 这将

给我们可以存储/读取该记录的服务器编号。这方法应确保服务器之间数据的统一分配。这种方法的根本问题是它有效地修复了总的数据库服务器的数量,因为添加新服务器意味着更改哈希值需要重新分配数据和停机时间的功能服务。此问题的解决方法是使用一致性哈希。

b.列表分区:在此方案中,每个分区都分配有一个值列表,所以每当我们想要插入一条新记录时,我们都会看到哪个分区包含我们的密钥,然后将其存储在那里。例如,我们可以决定所有用户居住在冰岛、挪威、瑞典、芬兰或丹麦的人将被存储在北欧国家的分治。

C.循环分区:这是一个非常简单的策略,可确保均匀的数据分布。对于“n”个分区,“i”元组被分配给分区 $(i \bmod n)$ 。

d.复合分区:在这种方案下,我们结合以上任何一种划分方案来设计新的方案。例如,首先应用列表分区方案,然后是基于哈希的分区。持续的散列可以被认为是散列和列表分区的组合,其中哈希将密钥空间减少到可以列出的大小。

3. 分片常见问题

在分片数据库上,不同的数据有一定的额外约束可以执行的操作。这些限制大部分是由于事实上,跨多个表或同一个表中的多行的操作将不再在同一服务器上运行。以下是一些限制和分片带来的额外复杂性:

A。连接和反规范化 :在正在运行的数据库上执行连接

在一台服务器上很简单,但是一旦数据库被分区并且

分布在多台机器上,执行以下连接通常是不可行的

跨数据库分片。此类连接的性能效率不高,因为数据

必须从多个服务器编译。一个常见的解决方法

问题是将数据库进行非规范化,以便查询以前的数据

可以从单个表执行所需的连接。当然还有服务

现在必须处理非规范化的所有危险,例如数据

不一致。

b. 参照完整性 :正如我们所看到的 ,在a上执行跨分片查询

分区数据库是不可行的,同样,尝试强制数据完整性

分片数据库中的外键等约束可能非常严重

困难的。

大多数 RDBMS 不支持跨数据库的外键约束

不同的数据库服务器。这意味着需要的应用程序

分片数据库上的引用完整性通常必须强制执行

应用程序代码。通常在这种情况下,应用程序必须运行常规 SQL

清理悬空引用的作业。

C。重新平衡 :我们必须改变分片的原因可能有很多

方案:

1. 数据分布不均匀,比如有很多地方可以放

无法放入一个数据库分区的特定邮政编码。

2. 分片负载过大,例如请求过多

由专用于用户照片的数据库分片处理。

在这种情况下,我们要么必须创建更多的数据库分片,要么必须重新平衡

现有的分片,这意味着分区方案发生了变化,所有

现有数据移至新位置。执行此操作不会导致停机
是极其困难的。使用像基于目录的分区这样的方案
使重新平衡成为一种更令人愉快的体验,但代价是增加
系统的复杂性并产生新的单点故障（即
查找服务/数据库）。

索引

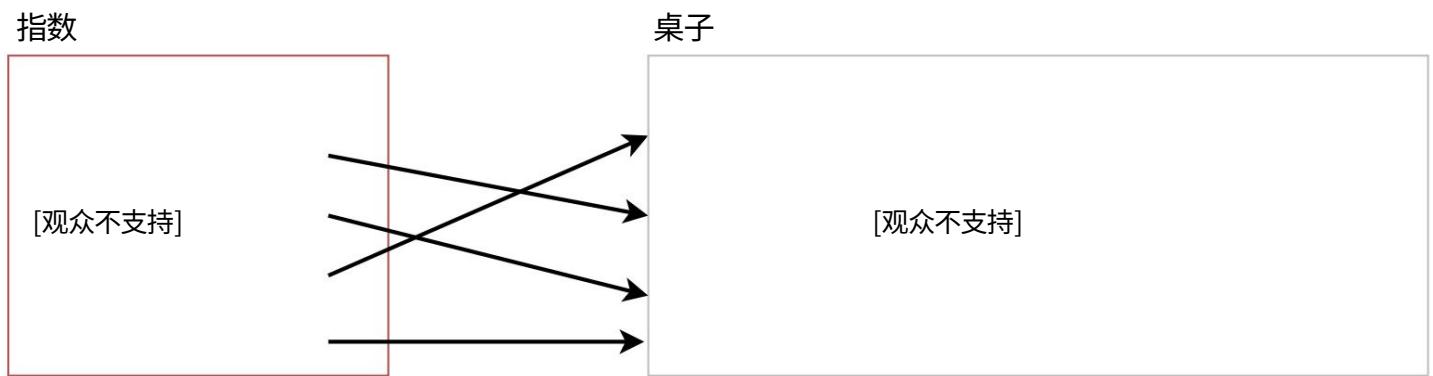
当谈到数据库时,索引是众所周知的。迟早会有
到了数据库性能不再令人满意的时候了。之一
当这种情况发生时,你首先应该求助的是数据库
索引。
在数据库中的特定表上创建索引的目的是使其
更快地搜索表并找到我们想要的行。
可以使用数据库表的一列或多列创建索引,
为快速随机查找和有效访问提供基础
有序记录。

示例:图书馆目录

图书馆目录是包含图书馆中找到的书籍列表的登记册。
该目录的组织方式类似于数据库表,通常有四列:
书名、作者、主题和出版日期。通常有两个这样的
目录:一份按书名排序,一份按作者姓名排序。
这样,你就可以想出你想读的作家,然后再看
浏览他们的书或查找您想要的特定书名
如果您不知道作者的名字,请阅读。这些目录就像

书籍数据库的索引。他们提供了一个排序的数据列表
可以轻松搜索到相关信息。

简单地说，索引是一种可以理解为表的数据结构
指向我们实际数据所在位置的内容。所以当我们
在表的列上创建索引，我们存储该列和一个指向
索引中的整行。让我们假设一个包含书籍列表的表，
下图显示了“标题”列上的索引的样子：



就像传统的关系数据存储一样，我们也可以将这个概念应用到
更大的数据集。索引的技巧在于我们必须仔细考虑如何
用户将访问数据。对于数 TB 的数据集
大小，但有效负载非常小（例如 1 KB），索引是必要的
优化数据访问。在如此大的数据集中找到一个小的有效负载可以
是一个真正的挑战，因为我们不可能迭代那么多数据
任何合理的时间。此外，如此大的数据集很可能
分布在多个物理设备上。这意味着我们需要某种方法
找到所需数据的正确物理位置。索引是最好的
方法来做到这一点。

索引如何降低写入性能?

索引可以显著加快数据检索速度,但由于索引本身可能很大
附加键,这会减慢数据插入和更新速度。
当为具有以下行的表添加行或更新现有行时
活动索引,我们不仅要写入数据还要更新
指数。这会降低写入性能。这次演出
降级适用于表的所有插入、更新和删除操作。
因此,应避免在表上添加不必要的索引
应删除不再使用的索引。重申一下,添加
索引旨在提高搜索查询的性能。如果目标是
数据库是提供一个经常写入而很少读取的数据存储
在这种情况下,降低更常见的性能
正在写入的操作可能不值得提高性能
我们从阅读中得到。
有关更多详细信息,请参阅[数据库索引](#)。

代理

代理服务器是客户端和后端之间的中间服务器
服务器。客户端连接到代理服务器以请求 Web 等服务
页面、文件、连接等,简单来说就是[代理服务器](#)是一个软件或
充当客户请求中介的硬件
来自其他服务器的资源。
通常,代理用于过滤请求、记录请求,或者有时
转换请求 (通过添加/删除标头、加密/解密或
压缩资源)。代理服务器的另一个优点是

缓存可以服务很多请求。如果多个客户端访问特定的资源,代理服务器可以缓存它并将其提供给所有客户端,而无需前往远程服务器。

代理服务器类型

代理可以驻留在客户端的本地服务器上或客户端之间的任何位置和远程服务器。以下是一些著名的代理服务器类型：

开放代理

[开放](#)代理是任何互联网用户都可以访问的代理服务器。

一般来说,代理服务器只允许某个网络组内的用户 (即封闭代理)来存储和转发 Internet 服务,例如 DNS 或 Web 页面以减少和控制组使用的带宽。与一个开放的代理,但是 Internet 上的任何用户都可以使用此转发服务。有两种著名的开放代理类型：

1.匿名代理- 该代理暴露了其作为服务器的身份,但确实

不透露初始 IP 地址。虽然这个代理服务器可以很容易发现,因为它隐藏了他们的 IP,因此对某些用户来说可能是有益的地址。

2.透明代理- 该代理服务器再次识别自己,并与 HTTP headers 的支持,可以查看第一个IP地址。这

使用此类服务器的主要好处是它能够缓存网站。

反向代理

反向代理代表客户端从一个或多个检索资源服务器。然后这些资源被返回给客户端,看起来就像它们一样源自代理服务器本身

冗余和复制

冗余是系统关键组件或功能的重复为了提高系统的可靠性,通常在形式的备份或故障安全,或提高实际系统性能。例如,如果一台服务器上仅存储一个文件的副本,则失去该服务器意味着失去文件。由于丢失数据很少是一件好事事情,我们可以创建文件的重复或冗余副本解决这个问题问题。

冗余在消除单点故障方面发挥着关键作用系统并在危机需要时提供备份。例如,如果我们有生产中运行的服务的两个实例,其中一个失败,系统可以故障转移到另一台。

复制意味着共享信息以确保之间的一致性冗余资源,例如软件或硬件组件,以改善可靠性、容错性、或可访问性。

复制广泛应用于许多数据库管理系统 (DBMS)中,通常,原件与被件之间存在主从关系副本。主站获取所有更新,然后波及到主站

奴隶。每个从站都会输出一条消息 ,表明它已收到更新成功,从而允许发送后续更新。

SQL 与 NoSQL

在数据库领域,有两种主要类型的解决方案 :SQL 和 NoSQL (或关系数据库和非关系数据库) 。两个都不同之处在于它们的构建方式、存储的信息类型以及他们使用的存储方法。

关系数据库是结构化的,并且具有预定义的模式,例如电话存储电话号码和地址的书籍。非关系型数据库是非结构化、分布式并且具有动态模式,例如文件夹保存从一个人的地址和电话号码到他们的所有信息 Facebook “点赞”和在线购物偏好。

SQL

关系数据库以行和列的形式存储数据。每行包含所有有关一个实体的信息,每一列包含所有单独的数据点。一些最流行的关系数据库是 MySQL、Oracle、MS SQL Server、SQLite、Postgres 和 MariaDB。

NoSQL

以下是最常见的 NoSQL 类型:

键值存储:数据存储在键值对数组中。“钥匙”是一个链接到“值”的属性名称。著名的键值存储包括 Redis、Voldemort 和 Dynamo。

文档数据库:在这些数据库中,数据存储在文档中

(而不是表中的行和列)并且这些文档被分组

一起收藏。每个文档可以有完全不同的

结构。文档数据库包括CouchDB和MongoDB。

宽列数据库:在列式数据库中,我们有而不是“表”

列族,是行的容器。与关系数据库不同,

我们不需要知道前面的所有列,也不需要知道每一行

具有相同的列数。列式数据库最适合

分析大型数据集 大名鼎鼎的包括 Cassandra 和 HBase。

图数据库:这些数据库用于存储其关系的数据

最好用图表来表示。数据以带有节点的图结构保存

(实体)、属性(有关实体的信息)和线(连接

实体之间)。图数据库的示例包括 Neo4J 和

无限图。

SQL 和 NoSQL 之间的高级差异

存储: SQL 将数据存储在表中,其中每一行代表一个实体,

每列代表有关该实体的一个数据点;例如,如果我们是

将汽车实体存储在表中,不同的列可以是“颜色”、“品牌”、

‘模型’ 等等。

NoSQL 数据库有不同的数据存储模型。主要有

键值、文档、图表和柱状。我们将讨论差异

下面这些数据库之间。

架构:在 SQL 中,每条记录都符合固定的架构,这意味着

必须在数据输入之前决定和选择列,并且每一行必须

每列都有数据。架构可以稍后更改,但它涉及

修改整个数据库并离线。

在 NoSQL 中,模式是动态的。可以动态添加列,每个列

“行” (或等效项)不必包含每个 “列”的数据。

查询: SQL数据库使用SQL (结构化查询语言)来定义

并操纵数据,这是非常强大的。在 NoSQL 数据库中,

查询集中于文档集合。有时也称为

UnQL (非结构化查询语言)。不同的数据库有不同的

使用 UnQL 的语法。

可扩展性:在大多数常见情况下,SQL 数据库是垂直可扩展的,

即,通过增加马力 (更高的内存、CPU 等)

硬件,这可能会变得非常昂贵。可以缩放关系

数据库跨多个服务器,但这是一个具有挑战性和时间

消耗过程。

另一方面,NoSQL 数据库是水平可扩展的,这意味着我们

可以在我们的 NoSQL 数据库基础设施中轻松添加更多服务器来处理

大量的流量。任何廉价的商品硬件或云实例都可以托管

NoSQL 数据库,因此比垂直数据库更具成本效益

缩放。许多 NoSQL 技术还跨服务器分布数据

自动地。

可靠性或 ACID 合规性 (原子性、一致性、隔离性、

持久性) :绝大多数关系数据库都符合 ACID。所以,

当涉及到数据可靠性和执行安全保障时

事务方面,SQL 数据库仍然是更好的选择。

大多数 NoSQL 解决方案会为了性能而牺牲 ACID 合规性

可扩展性。

SQL VS。 NoSQL - 使用哪一个？

就数据库技术而言,不存在一刀切的解决方案。

这就是为什么许多企业同时依赖关系型和非关系型

满足不同需求的数据库。即使 NoSQL 数据库正在崛起

由于其速度和可扩展性而受欢迎,但仍然存在以下情况:

高度结构化的 SQL 数据库可能性能更好;选择正确的

技术取决于用例。

使用 SQL 数据库的理由

以下是选择 SQL 数据库的几个原因:

1. 我们需要确保ACID合规性。 ACID 合规性降低

异常并通过规定保护数据库的完整性

事务如何与数据库交互。一般来说,NoSQL

数据库为了可扩展性和处理而牺牲了 ACID 合规性

速度,但对于许多电子商务和金融应用程序来说,ACID

兼容的数据库仍然是首选。

2. 您的数据是结构化且不变的。如果您的企业不是

经历大规模增长,需要更多服务器,如果

你只使用一致的数据,那么可能就没有

使用旨在支持多种数据类型的原因以及

高流量。

使用NoSQL数据库的理由

当我们应用程序的所有其他组件都快速且无缝时,

NoSQL 数据库可防止数据成为瓶颈。大数据是

为 NoSQL 数据库的巨大成功做出了贡献,主要是因为它

处理数据的方式与传统关系数据库不同。一些

NoSQL 数据库的流行示例包括 MongoDB、CouchDB、Cassandra、
和 HBase。

1. 存储大量几乎没有结构的数据。 A

NoSQL 数据库对我们可以存储的数据类型没有限制
在一起，并允许我们根据需求的变化添加新类型。和
基于文档的数据库，您可以将数据存储在一个地方，而无需
必须提前定义这些数据的“类型”。

2. 充分利用云计算和存储。基于云的存储

是一个优秀的节省成本的解决方案，但要求数据易于传播
跨多个服务器进行扩展。使用商品（价格实惠，
更小的）现场或云端硬件为您省去了以下麻烦
设计了额外的软件和 NoSQL 数据库（例如 Cassandra）
开箱即可跨多个数据中心扩展，无需大量
头痛。

3. 发展迅速。 NoSQL 对于快速开发非常有用

因为它不需要提前准备。如果您正在从事
系统的快速迭代需要频繁更新
到数据结构，而版本之间没有大量停机时间，
关系数据库会减慢你的速度。

CAP定理

CAP 定理指出，分布式软件系统不可能
同时提供以下三分之二以上
保证（CAP）：一致性、可用性和分区容错性。什么时候我们

设计一个分布式系统,CAP之间的权衡几乎是第一件事

我们要考虑。设计分布式系统时CAP定理的说法

我们只能选择以下三个选项中的两个：

一致性 :所有节点同时看到相同的数据。一致性是

通过在允许进一步读取之前更新多个节点来实现。

可用性 :每个请求都会获得成功/失败的响应。可用性是

通过在不同服务器之间复制数据来实现。

分区容错性 :尽管消息丢失或丢失,系统仍继续工作

部分失败。分区容忍的系统可以承受任意数量的

网络故障不会导致整个网络故障。数据是

跨节点和网络的组合充分复制,以保持

系统通过间歇性中断恢复运行。

我们无法建立一个持续可用的通用数据存储,

顺序一致,并且能够容忍任何分区故障。我们只能

构建一个具有这三个属性中任意两个的系统。因为,要成为一致,所有节点应该以相同的顺序看到相同的更新集。但是,如果网络出现分区,则某个分区中的更新可能无法进行在客户端从过期分区读取数据之前将其转移到其他分区读完最新的一篇后。唯一能做的就是应对这种可能性的方法是停止服务来自过时的请求分区,但服务不再 100% 可用。

一致性哈希

分布式哈希表 (DHT) 是分布式计算中使用的基本组件之一分布式可扩展系统。哈希表需要一个键、一个值和一个哈希值函数,其中哈希函数将键映射到值所在的位置存储。

索引 = hash_function(key)

假设我们正在设计一个分布式缓存系统。给定 n 缓存服务器,直观的哈希函数是 “ $\text{key \% } n$ ” 。它很简单并且常用。但它有两个主要缺点:

1. 它不可水平扩展。每当添加新的缓存主机时

系统中,所有现有的映射都被破坏。这将是一个痛点

如果缓存系统包含大量数据,则需要进行维护。实际上,它

安排停机时间来更新所有缓存变得很困难

映射。

2. 可能无法实现负载均衡,特别是对于非均匀分布的情况

数据。在实践中,可以很容易地假设数据不会被

均匀分布。对于缓存系统来说,它转化为一些

缓存变热并饱和,而其他缓存则空闲并处于空闲状态
几乎是空的。

在这种情况下,一致性哈希是改善缓存的好方法
系统。

什么是一致性哈希?

一致性哈希对于分布式缓存系统来说是一个非常有用策略
和 DHT。它允许我们以这样的方式跨集群分布数据:
添加或删除节点时将最大限度地减少重组。因此,
缓存系统将更容易扩展或缩小。

在一致性哈希中,当调整哈希表大小时(例如新的缓存主机
添加到系统中),仅需要重新映射“ k/n ”键,其中“ k ”是
键的总数,“ n ”是服务器的总数。回想一下,在一个
使用“mod”作为哈希函数的缓存系统,所有密钥都需要
重新映射。

在一致性哈希中,如果可能的话,对象会映射到同一主机。
当主机从系统中删除时,该主机上的对象将被共享
由其他主机;当添加新主机时,它会从一些主机中获取其份额
不碰别人的股份。

它是如何工作的?

作为典型的哈希函数,一致性哈希将键映射到整数。
假设哈希函数的输出在[0, 256)范围内。想象
将范围内的整数放在一个环上,使得这些值是
包起来。

以下是一致性哈希的工作原理:

1. 给定一个缓存服务器列表,将它们哈希为范围内的整数。

2. 要将密钥映射到服务器,

- 将其哈希为单个整数。
 - 在环上顺时针移动,直到找到第一个缓存遭遇。
 - 该缓存是包含密钥的缓存。请参阅下面的动画
- 举例 :key1 映射到缓存 A; key2 映射到缓存 C。

1共 5 个

要添加新服务器 (例如 D) ,原来驻留在 C 的密钥将被分裂。其中一些键将被移至 D,而其他键则不会被触摸。

要删除缓存,或者如果缓存失败,例如 A,则删除原来的所有密钥映射到A的键会落入B,只有那些键需要移动到B;
其他按键不会受到影响。

对于负载平衡,正如我们在开始时讨论的那样,真实数据是本质上是随机分布的,因此可能不均匀。它可能使缓存上的键不平衡。

为了解决这个问题,我们为缓存添加了“虚拟副本”。而不是映射每个缓存到环上的单个点,我们将其映射到环上的多个点戒指,即复制品。这样,每个缓存都与多个部分相关联戒指。

如果哈希函数“混合得很好”,随着副本数量的增加,按键会更加平衡。

长轮询 vs WebSockets vs 服务器 已发送事件

长轮询、WebSocket 和服务器之间有什么区别

发送事件？

长轮询、WebSocket 和服务器发送事件很流行

客户端（例如 Web 浏览器）和 Web 之间的通信协议

服务器。首先，我们先来了解一下什么是标准的 HTTP Web 请求

好像。以下是常规 HTTP 请求的一系列事件：

1. 客户端打开连接并向服务器请求数据。

2. 服务器计算响应。

3. 服务器将打开的响应发送回客户端

要求。



阿贾克斯轮询

轮询是绝大多数 AJAX 使用的标准技术

应用程序。基本思想是客户端重复轮询（或请求）

数据服务器。客户端发出请求并等待服务器

用数据回应。如果没有可用数据，则返回空响应。

1. 客户端打开连接并向服务器请求数据

常规 HTTP。

2. 被请求的网页定时向服务器发送请求

间隔（例如0.5秒）。

3. 服务器计算响应并将其发送回来，就像常规一样

HTTP 流量。

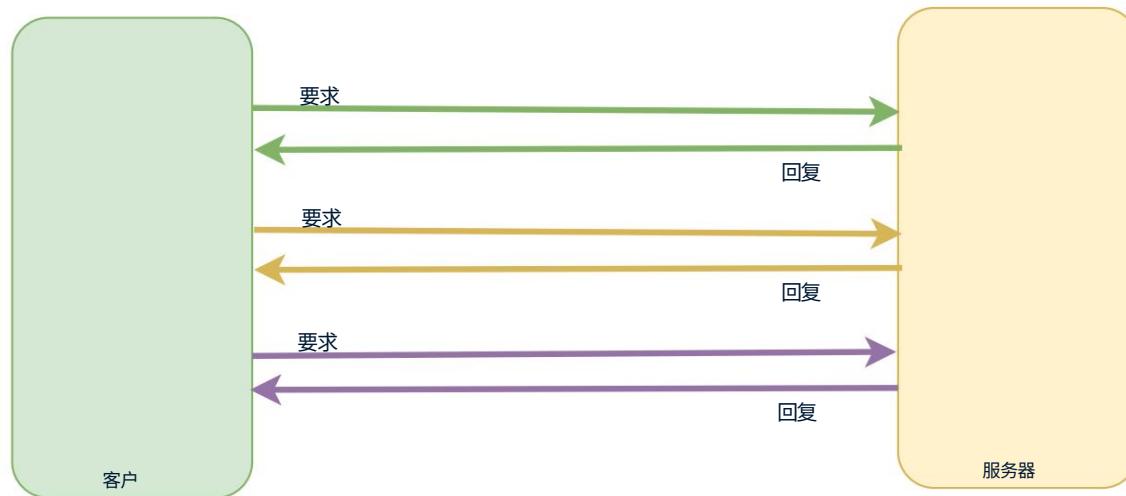
4. 客户端周期性地重复以上三个步骤来获取更新

从服务器。

轮询的问题是客户端必须不断向服务器询问

任何新数据。结果，很多响应都是空的，创建了 HTTP

高架。



Ajax 轮询协议

HTTP 长轮询

这是传统轮询技术的一种变体,允许服务器

只要数据可用,就将信息推送给客户端。与长

轮询,客户端像平常一样向服务器请求信息

轮询,但期望服务器可能不会响应

立即地。这就是为什么该技术有时被称为

“**悬挂GET**”。

· 如果服务器没有任何可供客户端使用的数据,则改为

发送空响应,服务器保留请求并等待

直到某些数据可用。

· 一旦数据可用,就会向客户端发送完整的响应。

然后客户端立即向服务器重新请求信息,以便

服务器几乎总是有可用的等待请求

它可以用来自传递数据以响应事件。

使用 HTTP Long-Polling 的应用程序的基本生命周期如下:

1. 客户端使用常规 HTTP 发出初始请求,然后等待

以获得回应。

2. 服务器延迟响应,直到有更新可用或超时

已经发生了。

3. 当有更新可用时,服务器向更新发送完整响应

客户。

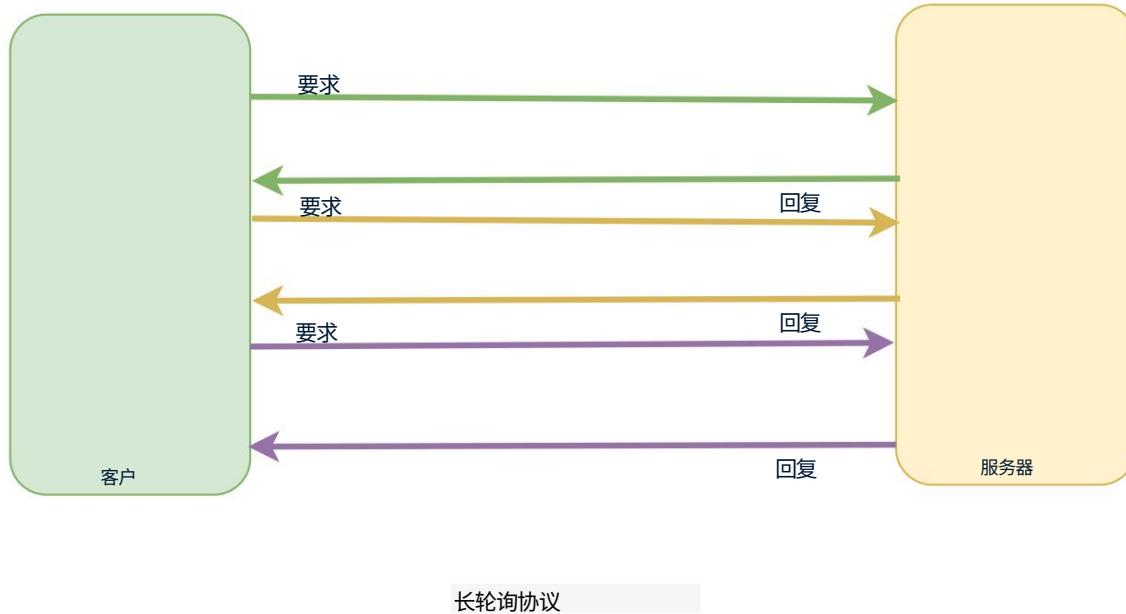
4. 客户端通常会发送一个新的长轮询请求,或者立即发送

收到响应后或暂停后允许可接受的

潜伏期。

5. 每个长轮询请求都有超时时间。客户端必须重新连接

由于超时而关闭连接后定期进行。



WebSockets

WebSocket 提供[全双工](#)沟通渠道

单个 TCP 连接。它提供了客户端之间的持久连接

以及双方可以随时开始发送数据的服务器。这

客户端通过一个称为 WebSocket 连接的进程建立 WebSocket 连接

WebSocket 握手。如果该过程成功，则服务器和客户端

可以随时双向交换数据。 WebSocket 协议

以较低的开销实现客户端和服务器之间的通信，

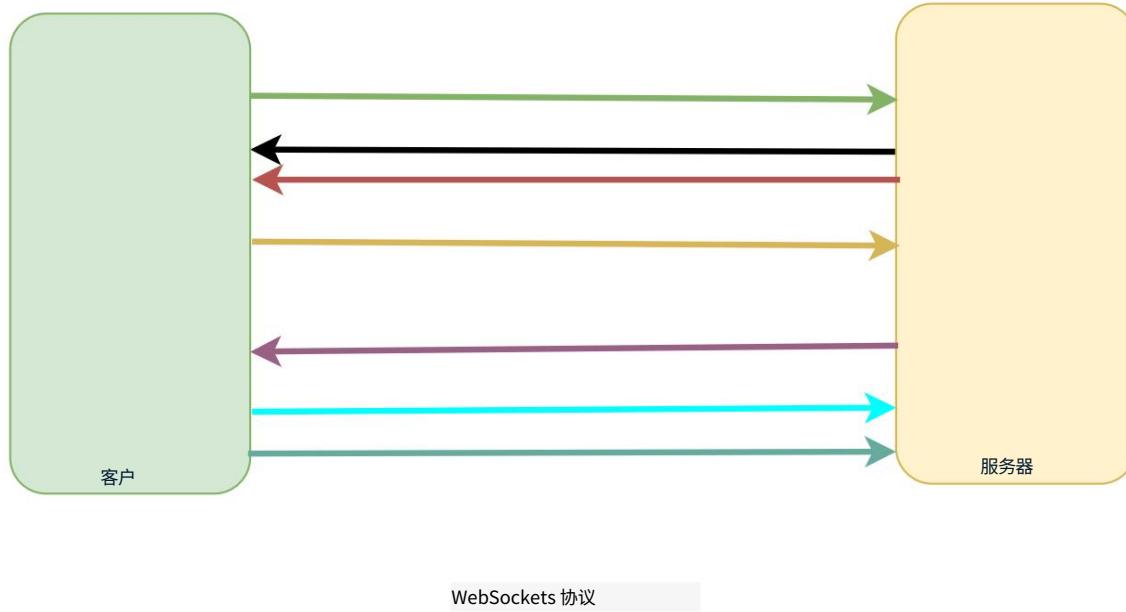
促进与服务器之间的实时数据传输。这是做的

通过为服务器提供标准化的方式将内容发送到

浏览器无需客户端询问并允许消息

在保持连接打开的同时来回传递。这样，一个

双向 (双向)正在进行的对话可以在客户端和服务器。



服务器发送的事件 (SSE)

在 SSE 下 ,客户端与以下设备建立持久且长期的连接：

服务器。服务器使用此连接将数据发送到客户端。如果

客户端想要向服务器发送数据,就需要使用另一个

这样做的技术/协议。

1.客户端使用常规HTTP从服务器请求数据。

2. 请求的网页打开与服务器的连接。

3.服务器端有新数据时 ,将数据发送给客户端

可用的信息。

当我们需要从服务器到客户端的实时流量或

如果服务器正在循环中生成数据并将发送多个事件

给客户。

