Simple URL Shortening Services

System requirements

- **Functional:** Ability to take the URL and return a shorter format
- **Non-Functional:**
  1. Availability over consistency
  2. Low latency: tiny URL provide minimum overhead最小开销
  word wide use case

Capacity estimation

Estimatons:

我们可以看到，我们每秒大约有 100 次读取请求，可能还有每秒 1 次写入请求。如果我们每秒有 1 次读取请求，要编码一个 URL 可能需要 8 字节的数据，假设一天有大约 10 * 10^5 秒，我们每天大约需要 8 * 10^5 字节的数据，即每天 800 MB 的数据。

考虑到每秒 100 次读取请求，我们不能仅依靠one host来处理所有请求，我们需要scale beyond that。

API Design

为了实现这个功能，我们需要设计两个 API，

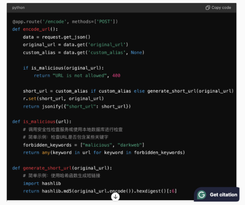
- 一个用于处理 GET 请求以重定向 URL，

- 另一个用于通过 UI 门户让用户编码他们的 URL 1

我们需要ensure encpded URL 不恶意aren't malicious or point to dark/ denied resources或指向被拒绝的资源。这可以通过使用安全性检查服务或数据库来实现。

Database Design

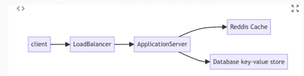
simple (persistent) key-value store --> NoSQL



- **Redis**：高性能的内存数据库，支持持久化。
- **Amazon DynamoDB**：完全托管的 NoSQL 键值和文档数据库，具有高可用性和自动扩展能力。
- **Cassandra**：分布式 NoSQL 数据库系统，擅长处理大规模数据。

High-level Design

Architecture: 客户端首先访问负载均衡器，负载均衡器会根据 RR stateless algorithm. 将请求重定向到合适的应用服务器。



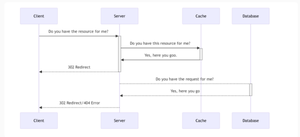
应用服务器会首先检查 URL 是否存在于内存缓存中（我们可以根据缓存客户端守护进程在应用服务器上运行时使用的一致性哈希来确定要检查哪个缓存服务器，缓存是 LRU（最近最少使用）算法）。

如果命中缓存，我们会直接返回结果。否则，我们会检查数据库（键值存储）。NoSQL 数据库能够将请求路由到正确的分区并返回请求的结果。如果没有找到，返回 404。

总体来说，我可以接受在资源添加后需要一段时间才能完全传播到所有副本（最终一致性是可以的）。然而，我们希望保持数据的持久性（确保不会丢失写请求）和低延迟（尽可能快地重定向 <20ms）。

Request Flow

Writhing flow: write-around cache, means write to DB not to cache。
The cache entries are updated on write flow to minimize read latency
缓存条目在写入流程中被更新，以最小化读取延迟（trading off write latency for read latency since the system has to be very fast）



Detailed Component design

数据中心内replicating Synchronously 同步复制，数据中心外异步复制 replicating synchronously within the datacenter + async outside the datacenter.

确保在同一个数据中心内的所有副本replicetes都保持一致consistent。

在数据中心之间异步复制数据 async outside the datacenter，允许一定的延迟latency。

版本冲突不是主要问题version conflicts are not the major problems

一个 URL 可以映射到两个不同的短链接，这不会影响系统功能。

分布式唯一 ID 生成器 Distributed Unique ID generator

使用哈希算法生成短链接。例如将 /Facebook/com/post-id/1231 转换为 aH102

Ensure different strings won't generate same hash value

After each hash generating check DB collisions

哈希算法示例

```python
import hashlib
import random

def generate_short_url(original_url):
    hash_object = hashlib.md5(original_url.encode())
    short_url = hash_object.hexdigest()[:6]
    while check_hash_conflict(short_url, original_url):
        # 如果存在冲突，重新生成
        hash_object.update(str(random.random()).encode())
        short_url = hash_object.hexdigest()[:6]
    return short_url

def check_hash_conflict(short_url, original_url):
    # 检查数据库中是否存在冲突
    existing_url = get_long_url_from_db(short_url)
    return existing_url is not None and existing_url != original_url
```

Trade-Offs/ Tech Choices

Trade-offs权衡 in Designing Read/Write Flows Around Cache。一些选择包括谁将数据添加到缓存中，谁只从缓存中读取。我们在这里权衡了写入延迟（允许较慢）和读取延迟（我们希望尽可能快）。

我们做出的另一个权衡涉及一致性。我们再次在一致性和延迟之间做出权衡，希望在不完全等待所有副本完成的情况下尽快提供读取请求。原因是我们关心延迟，并且短链接不需要在创建后的几秒钟内立即可用，因为用户可能不会立即分享链接。

读/写缓存流程的权衡Trade-off in Designing Read/Write Flows Around Cache

写入延迟 vs. 读取延迟：Slower Write Latency and faster Read Latency**:

一致性 vs. 延迟

一致性：

Data consistency requirements可以稍微降低要求。

延迟：

读取请求需要尽快响应，而不需要等待所有副本完成复制。Read requests need to respond quickly without waiting for all replicas to complete.

链接在创建后不需要立即可用，因为用户可能不会立即分享链接。Links do not need to be immediately available upon creation since users are unlikely to share them instantly.

Failure Scenarios/Bottlenecks

详细组件设计中的单点故障 (SPOF)

虽然我们可以部署多个负载均衡器、服务器、数据库节点和缓存节点，但需要非常小心生成短链接哈希的算法，因为有很多可能出错的地方：

- 哈希用尽后，短链接变得不再短。

  解决方案: 增加哈希算法的复杂性，使用更大的哈希空间（如 Base62 编码）以确保短链接足够短小。

- 我们在创建时进行了大量的数据库调用，降低了写入延迟。

  解决方案: 采用批量写入策略，减少单次写入操作的频率和数量。

- 我们没有有效地跟踪已使用的哈希。

  解决方案: 使用一个去重集合（例如 Redis Set）来存储所有已使用的哈希，并在生成新哈希时进行检查。

- 如果某个非常有名的人发布了一个 URL，会产生大量流量，系统可能会崩溃。

  解决方案: 采用负载均衡和弹性扩展策略，确保系统能够动态扩展以应对突发流量。

- 有几个组件引入了单点故障（SPOF）。

  解决方案: 为所有关键组件（如负载均衡器、服务器、数据库节点和缓存节点）设置冗余和故障转移机制。

Future Improvements