Tiny URL

1)Short links save a lot of space when displayed, printed, messaged, or tweeted. Additionally, users are less likely to mistype shorter URLs

2)The shortened URL is nearly one-third the size of the actual URL.

3)URL shortening is used for optimizing links across devices, tracking individual links to analyze audience and campaign performance, and hiding affiliated original URLs.

4)Functional Requirements:

1. Given a URL, our service should generate a shorter and unique alias of it. This is called a short link. This link should be short enough to be easily copied and pasted into applications.
2. When users access a short link, our service should redirect them to the original link.
3. Users should optionally be able to pick a custom short link for their URL.
4. Links will expire after a standard default timespan. Users should be able to specify the expiration time.

5)Non-Functional Requirements:

1. The system should be highly available. This is required because, if our service is down, all the URL redirections will start failing.
2. URL redirection should happen in real-time with minimal latency.
3. Shortened links should not be guessable (not predictable).

6)Extended Requirements

1. Analytics; e.g., how many times a redirection happened?
2. Our service should also be accessible through REST APIs by other services.

7) Our system will be read-heavy. There will be lots of redirection requests compared to new URL shortenings. Let’s assume a 100:1 ratio between read and write.

Back of the envelope Calculations:

Traffic Estimate :

500M new URL shortenings per month, with 100:1 read/write ratio, we can expect 50B redirections during the same period:

100 \* 500M => 50B

What would be Queries Per Second (QPS) for our system? New URLs shortenings per second:

500 million / (30 days \* 24 hours \* 3600 seconds) = ~200 URLs/s

Considering 100:1 read/write ratio, URLs redirections per second will be:

100 \* 200 URLs/s = 20K/s

**Storage estimates:** Let’s assume we store every URL shortening request (and associated shortened link) for 5 years. Since we expect to have 500M new URLs every month, the total number of objects we expect to store will be 30 billion:

500 million \* 5 years \* 12 months = 30 billion

Let’s assume that each stored object will be approximately 500 bytes (just a ballpark estimate–we will dig into it later). We will need 15TB of total storage:

30 billion \* 500 bytes = 15 TB

**Bandwidth estimates:** For write requests, since we expect 200 new URLs every second, total incoming data for our service will be 100KB per second:

200 \* 500 bytes = 100 KB/s

For read requests, since every second we expect ~20K URLs redirections, total outgoing data for our service would be 10MB per second:

20K \* 500 bytes = ~10 MB/s

**Memory estimates:** If we want to cache some of the hot URLs that are frequently accessed, how much memory will we need to store them? If we follow the 80-20 rule, meaning 20% of URLs generate 80% of traffic, we would like to cache these 20% hot URLs.

Since we have 20K requests per second, we will be getting 1.7 billion requests per day:

20K \* 3600 seconds \* 24 hours = ~1.7 billion

To cache 20% of these requests, we will need 170GB of memory.

0.2 \* 1.7 billion \* 500 bytes = ~170GB

One thing to note here is that since there will be a lot of duplicate requests (of the same URL), therefore, our actual memory usage will be less than 170GB.

**High level estimates:** Assuming 500 million new URLs per month and 100:1 read:write ratio, following is the summary of the high level estimates for our service:

|  |  |
| --- | --- |
| New URLs | 200/s |
| URL redirections | 20K/s |
| Incoming data | 100KB/s |
| Outgoing data | 10MB/s |
| Storage for 5 years | 15TB |
| Memory for cache | 170GB |

System APIs :

We can have SOAP or REST APIs to expose the functionality of our service. Following could be the definitions of the APIs for creating and deleting URLs:

createURL(api\_dev\_key, original\_url, custom\_alias=None, user\_name=None, expire\_date=None)

**Parameters:**api\_dev\_key (string): The API developer key of a registered account. This will be used to, among other things, throttle users based on their allocated quota.original\_url (string): Original URL to be shortened.custom\_alias (string): Optional custom key for the URL.user\_name (string): Optional user name to be used in the encoding.expire\_date (string): Optional expiration date for the shortened URL.

**Returns:** (string)A successful insertion returns the shortened URL; otherwise, it returns an error code.

deleteURL(api\_dev\_key, url\_key)

Where “url\_key” is a string representing the shortened URL to be retrieved. A successful deletion returns ‘URL Removed’.

Throttling part :

**How do we detect and prevent abuse?** A malicious user can put us out of business by consuming all URL keys in the current design. To prevent abuse, we can limit users via their api\_dev\_key. Each api\_dev\_key can be limited to a certain number of URL creations and redirections per some time period (which may be set to a different duration per developer key).

**Database Design**

💡     ***Defining the DB schema in the early stages of the interview would help to understand the data flow among various components and later would guide towards data partitioning.***

A few observations about the nature of the data we will store:

1. We need to store billions of records.
2. Each object we store is small (less than 1K).
3. There are no relationships between records—other than storing which user created a URL.
4. Our service is read-heavy.

**Database Schema:**

We would need two tables: one for storing information about the URL mappings, and one for the user’s data who created the short link.

**What kind of database should we use?** Since we anticipate storing billions of rows, and we don’t need to use relationships between objects – a NoSQL key-value store like [DynamoDB](https://en.wikipedia.org/wiki/Amazon_DynamoDB), [Cassandra](https://en.wikipedia.org/wiki/Apache_Cassandra) or [Riak](https://en.wikipedia.org/wiki/Riak) is a better choice.

We would need two tables: one for storing information about the URL mappings, and one for the user’s data who created the short link.

**a. Encoding actual URL**

We can compute a unique hash (e.g., [MD5](https://en.wikipedia.org/wiki/MD5) or [SHA256](https://en.wikipedia.org/wiki/SHA-2), etc.) of the given URL. The hash can then be encoded for displaying. This encoding could be base36 ([a-z ,0-9]) or base62 ([A-Z, a-z, 0-9]) and if we add ‘+’ and ‘/’ we can use [Base64](https://en.wikipedia.org/wiki/Base64#Base64_table) encoding. A reasonable question would be, what should be the length of the short key? 6, 8, or 10 characters.

Using base64 encoding, a 6 letters long key would result in 64^6 = ~68.7 billion possible strings

Using base64 encoding, an 8 letters long key would result in 64^8 = ~281 trillion possible strings

With 68.7B unique strings, let’s assume six letter keys would suffice for our system.

If we use the MD5 algorithm as our hash function, it’ll produce a 128-bit hash value. After base64 encoding, we’ll get a string having more than 21 characters (since each base64 character encodes 6 bits of the hash value). Since we only have space for 8 characters per short key, how will we choose our key then? We can take the first 6 (or 8) letters for the key. This could result in key duplication, to resolve that, we can choose some other characters out of the encoding string or swap some characters.

**What are the different issues with our solution?** We have the following couple of problems with our encoding scheme:

1. If multiple users enter the same URL, they can get the same shortened URL, which is not acceptable.
2. What if parts of the URL are URL-encoded? e.g., <http://www.educative.io/distributed.php?id=design>, and [http://www.educative.io/distributed.php%3Fid%3Ddesign](http://www.educative.io/distributed.php?id=design) are identical except for the URL encoding.

**Workaround for the issues:** We can append an increasing sequence number to each input URL to make it unique, and then generate a hash of it. We don’t need to store this sequence number in the databases, though. Possible problems with this approach could be an ever-increasing sequence number. Can it overflow? Appending an increasing sequence number will also impact the performance of the service.

Another solution could be to append user id (which should be unique) to the input URL. However, if the user has not signed in, we would have to ask the user to choose a uniqueness key. Even after this, if we have a conflict, we have to keep generating a key until we get a unique one.

For simplicity, as soon as KGS loads some keys in memory, it can move them to the used keys table. This ensures each server gets unique keys.

KGS keeps keys in memory.

Load some keys in memory and rest of them on disk.

Implements concurrency control via locking to avoid giving same key to two servers twice.

We can even combine certain less frequently occurring letters into one database partition. We should come up with a static partitioning scheme so that we can always store/find a URL in a predictable manner.

In our case, we can take the hash of the ‘key’ or the short link to determine the partition in which we store the data object.

Cache 20% of daily traffic -  Hot urls -  short url to Main Url Map.

Another benefit of this approach is that if a server is dead, LB will take it out of the rotation and will stop sending any traffic to it.

Instead, we can slowly remove expired links and do a lazy cleanup

This service should be very lightweight and can be scheduled to run only when the user traffic is expected to be low.

Should we remove links that haven’t been visited in some length of time, say six months? This could be tricky. Since storage is getting cheap, we can decide to keep links forever.

Given that we are storing our data in a NoSQL wide-column database like Cassandra, the key for the table storing permissions would be the ‘Hash’ (or the KGS generated ‘key’). The columns will store the UserIDs of those users that have the permission to see the URL.