Designing a URL Shortening Service like TinyURL

Difficulty Level: Easy

Let’s design a URL shortening service like TinyURL.

This service will provide short aliases redirecting to long URLs.

1. Why do we need URL shortening?

URL shortening is used to create shorter aliases for long URLs.

We call these shortened aliases “short links”.

Users are redirected to the original URL when they hit these short links.

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

URL shortening is used to optimise links across devices, track individual links to analyse audience, measure ad campaign performance, or hide affiliated original URLs.

2. Requirements and Goals of the System

❗ We should always clarify requirements at the beginning of the interview.

Be sure to ask questions to find the exact scope of the system that the interviewer has in mind.

Our URL shortening system should meet the following requirements:

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.

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).

Extended Requirements

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

3. Capacity Estimation and Constraints

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.

Traffic Estimates: Assuming that we will have 500M new URL shortenings per month, with a 100:1 read-write ratio, we can expect 50B redirections during the same period:

$$500M * 100 = 50B \text{ redirections}$$

What would be the Queries Per Second for our system? New URL shortenings per second:

$$500M/(30 \text{ days} \cdot 24 \text{ hours} \cdot 3600 \text{ seconds}) \approx 200 \text{ URLs per second}$$

Considering a 100:1 read-write ratio, URL redirections per second will be:

$$100 \cdot 200 \text{ URLs per second} = 20 \text{ K/s}$$

Storage Estimates: Let’s assume that 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:

$$500M \cdot 5 \text{ years} \cdot 12 \text{ months} = 30B \text{ objects}$$

Let’s assume that each stored object will be approximately 500 bytes. This means that we will need 15TB of total storage:

$$30B \text{ objects} \cdot 500 \text{ bytes per object} = 15TB$$

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

$$200 \cdot 500 \text{ bytes} = 100 \text{ KB/s}$$

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

$$20K \cdot 500 \text{ bytes} \approx 10 \text{ MB/s}$$

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 the 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:

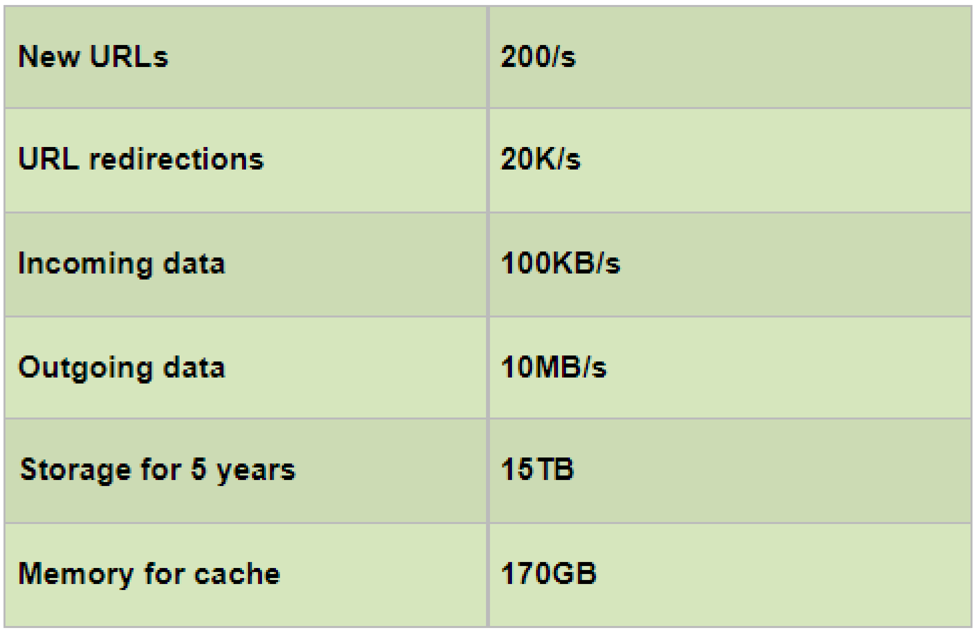
$$20K \cdot 3600 \text{ seconds} \cdot 24 \text{ hours} \approx 1.7B$$

Then, to cache 20% of these requests, we will need 170GB of memory:

$$0.2 \cdot 1.7B \cdot 500 \text{ bytes} \approx 170 \text{ GB}$$

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

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



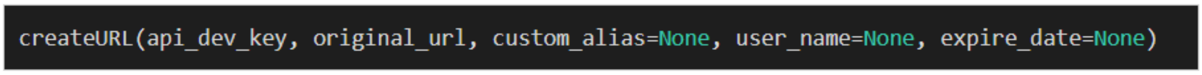
New URLs	200/s
URL redirections	20K/s
Incoming data	100KB/s
Outgoing data	10MB/s
Storage for 5 years	15TB
Memory for cache	170GB

4. System APIs

> ❗ Once we've finalised the requirements, it's always a good idea to define the system APIs.
>
> This should explicitly state what is expected from the system.

We can have SOAP or REST APIs to expose the functionality of our service.

The 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) — the original URL to be shortened

- `custom_alias` (string) — optional custom key for the URL
- `username` (string) — optional username 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)
```

Parameters:

- `api_dev_key` (string) — same as above
- `url_key` (string) — the shortened URL to be deleted

Returns: (string)

A successful deletion returns some success message; otherwise, it returns an error code and does not proceed with the deletion.

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 by their `api_dev_key`.

Each API 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.

5. Database Design

❗ Defining the database schema in the early stages of the interview will help to understand the data flow among various components and later will guide towards data partitioning.

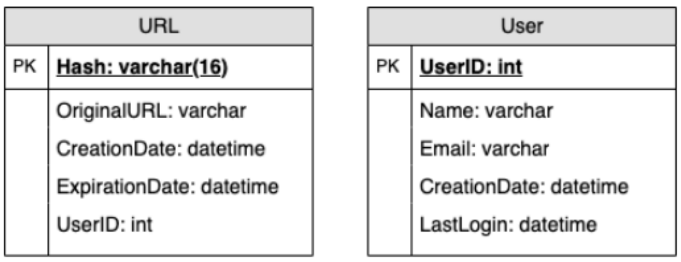
A few observations about the nature of the data that we will store:

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



	URL
PK	**Hash: varchar(16)**
	OriginalURL: varchar
	CreationDate: datetime
	ExpirationDate: datetime
	UserID: int

	User
PK	**UserID: int**
	Name: varchar
	Email: varchar
	CreationDate: datetime
	LastLogin: datetime

URL shortening DB schema

What kind of database should we use?

Since we anticpate storing billions of row,s and we don't need to use relationships between objects, a NoSQL store like Cassandra or MongoDB is a good choice of database to use.

A NoSQL choice would also be easier to scale.

6. Basic System Design and Algorithm

The problem that we are solving here is how to generate a short and unique key for a given URL.

Take the shortened URL `https://tinyurl.com/vzet59pa` — the last 8 characters of this URL constitute that short key that we want to generate.

We'll explore two solutions here.

a. Encoding the actual URL

We can compute a unique hash (e.g. MD5 or SHA256 etc.) of the given URL.

The hash can then be encoded for display. The encoding could be Base36 ([a-z0-9]), Base62 ([a-zA-Z0-9]) or Base64 if we also use ‘+’ and ‘/’.

A reasonable question then is what the length of the short key should be: 6, 8, or 10 characters?

Using Base64 encoding, a 6 letter long key would result in $64^6 \approx 68.7$ billion possible strings.

Using Base64 encoding, a 8 letter long key would result in $64^8 \approx 281$ trillion possible strings.

With 68.7B unique strings, let’s assume that 6 letter keys would suffice for our system.

If we use the MD5 algorithm as our hash function, it will 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 6 (or 8) characters per short key, we need to decide how to choose our key.

We could take the first 6 (or 8) letters for the key, but this could result in key duplication. To resolve this, we could choose some other characters out of the encoding string or swap some characters.

What are the different issues with our solution?

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.designgurus.org/distributed.php?id=design and http://www.designgurus.org /distributed.php%3Fid%3Ddesign are identical except for the URL encoding

Workaround for these issues

We can append an increasing sequence number to each input URL to make it unique, and then generate its hash. We don't need to store this sequence number in the database, we can just retrieve it from the number of records in the database itself.

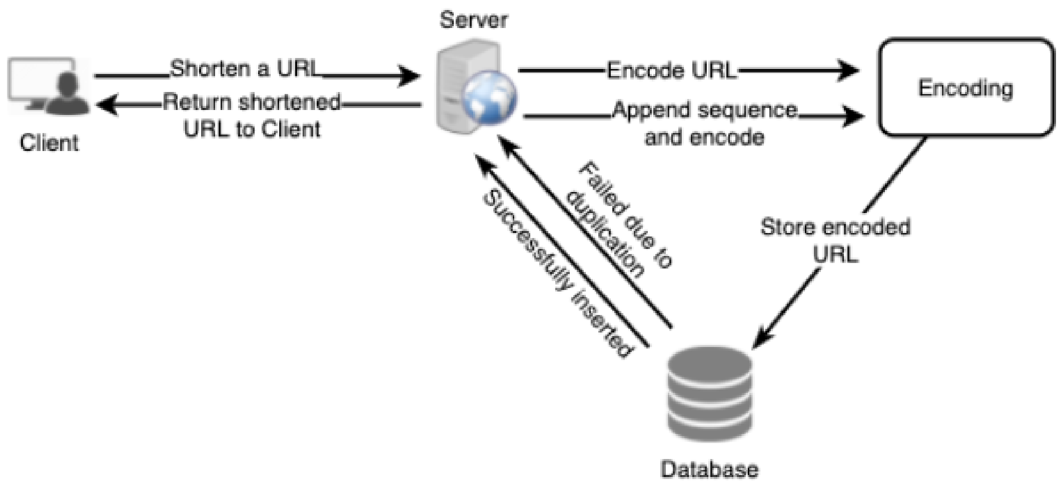
Possible problems with this approach:

- An ever-increasing sequence number. Can it overflow?
- Negatively impacts the performance of the service

Another solution could be to append the 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 will have to keep generating a key until we get a unique one.



Request flow for shortening of a URL

b. Generating keys offline

We could have a standalone Key Generation Service (KGS) that generates random six-letter strings beforehand, and stores them in a database (let's call it key-DB).

Then, whenever we want to shorten a URL, we will take one of the already generated keys and use it.

This approach will make things quite simple and fast.

Not only are we not encoding the URL, but we won't have to worry about duplications or collisions. KGS will make sure al the keys inserted into key-DB are unique.

Can concurrency cause problems?

As soon as a key is used, it should be marked in the database to ensure that is is not used again.

If there are multiple servers reading keys concurrently, we might get a scenario where two or more servers try to read the same key from the database.

So, how do we solve this concurrency problem?

Servers can use KGS to read/mark keys in the database. KGS can use two tables to store keys: one for keys that are not used yet, and one for all the used keys.

As soon as KHS gives keys to one of the servers, it can move them to the used keys table.

KGS can always keep some keys in memory to quickly provide them whenever a server needs them.

For simplicity, as soon as KHS loads some keys in memory, it can move them to the used keys table.

This ensures that each server gets unique keys.

If KGS dies before assigning all the loaded keys to some server, we will be wasting those keys — this could be acceptable given the huge number of keys that we have.

KGS also has to make sure not to give the same key to multiple serer.

For that, it must synchronise (or get a lock on) the data structure holding the keys before removing keys from it and giving them to a server.

What would be the size of key-DB?

With Base64 encoding, we can generate 68.7B unique six letter keys.

If we need one byte to store one alphanumeric character, we can store all these keys in:

$$6 \text{ characters per key} \cdot 68.7B \text{ unique keys} = 412 \text{ GB}$$

Isn’t KGS a single point of failure?

Yes, it is.

To solve this, we can have a standby replica of KGS.

Whenever the primary server dies, the standby server can take over to generate and provide keys.

Can each app server cache some keys from key-DB?

Yes, this would surely speed things up.

Although, in this case, if the application serer dies before consuming all the keys, we will end up losing those keys.

This can be acceptable since we have 68.7B unique 6-letter keys.

How would we perform a key lookup?

We can look up the key in our database (since its a primary key) to get the full URL.

If it is present in the DB, issue an “HTTP 302 Redirect” status back to the browser, passing the stored URL in the “Location” field of the response.

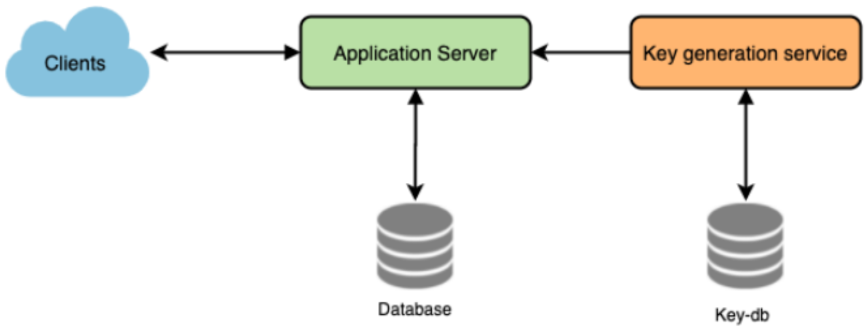
If that key is not present in our system, issue an “HTTP 404 Not Found” status or redirect the user back to the homepage.

Should we impose size limits on custom aliases?

Our service supports custom aliases. Users can pick any “key” they like, but providing a custom alias is not mandatory.

However, it is reasonable and desirable to impose a size limit on a custom alias to ensure that we have a consistent URL databse.

Let’s assume that users can specify a maximum of 16 characters per customer key (varchar(16)).



High level system design for URL shortening

7. Data Partitioning and Replication

To scale out our DB, we need to partition it so that it can store information about billions of URLS.

Therefore, we need to develop a partitioning schema that would divide and store our data into different DB servers.

a. Range-based Partitioning

We can store URLs in separate partitions based on the hash key’s first letter.

Hence, we will save all the URL hash keys starting with the letter ‘A’ or ‘a’ in one partition, and save those that start with the letter ‘B’ or ‘b’ in another partition, and so on.

This approach is called **range-based partitioning**.

We can combine certain less frequently occurring letters into one database partition.

Thus, we should develop a static partitioning scheme to always store/find a URL in a predictable manner.

The main problem with this approach is that it can lead to unbalanced DB servers.

For example, we decide to put all URLs starting the with the letter ‘E’ into a DB partition, but later we realise that we have too many URLs that start with the letter E.

b. Hash-based Partitioning

In this scheme, we take a hash of the object that we are storing.

We then calculate which partition to use based upon the hash. In our case, we can take the hash of the ‘key’ or the short link to determine the partition in which we will store the data object.

Our hashing function will randomly distribute the URLs into different partitions (e.g. our hashing function can always map any ‘key’ to a number between [1…256]).

This number would represent the partition in which we store our object.

This approach can still lead to overloaded partitions, which can be solved using Consistent Hashing.

8. Cache

We can cache URLs that are frequently accessed.

We can use any off-the-shelf-solution like Memcached, which can store full URLs with their respective ahshes.

Thus, the application servers, before hitting the backend storage, can quickly check if the cache has the desired URL.

How much cache memory should we have?

We can start with 20% of daily traffic and, based on clients’ usage patterns, we can adjust how many cache servers we need.

As estimated above, we need 170GB of memory to cache 20% of daily traffic.

Since a modern-day server can have 256GB of memory, we can easily fit all the cache into one machine.

Alternatively, we can use a few smaller servers to store all of these hot URLs.

Which cache eviction policy would best fit our needs?

When the cache is full, and we want to replace a link with a newer/hotter URL, how do we choose which link to evict?

Least Recently Used (LRU) can be a reasonable policy for our system.

Under this policy, we discard the least recently used URL first.

We can use a Linked HashMap or a similar data structure to store our URLs and hashes, which will also keep track of the URLs that have been accessed recently.

To further increase efficiency, we can replicate our caching servers to distribute the load between them

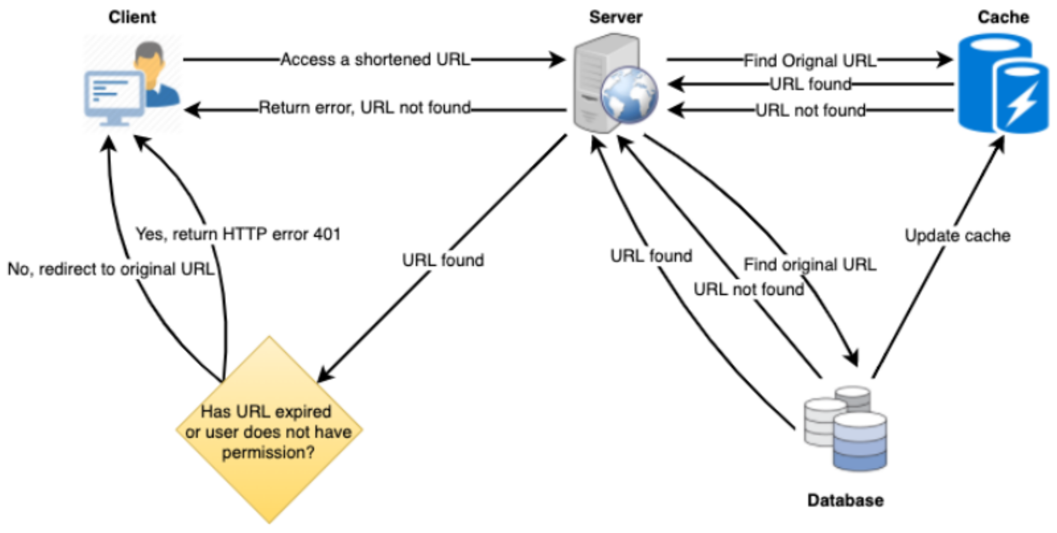
How can each cache replica be updated?

Whenever there is a cache miss, our servers would be hitting a backend database.

Whenever this happens, we can update the cache and pas the new entry to all the cache replicas.

Each replica can update its cache by adding the new entry.

If a replica already has that entry, it can simply ignore it.



Request flow for accessing a shortened URL

9. Load Balancer (LB)

We can add a **Load Balancing (LB)** layer at three places in our system:

1. Between the client and the application server

2. Between the application server and the database server
3. Between the application server and the cache server

Initially, we could use a simple Round Robin approach that distributes incoming requests equally among backend servers.

this LB is simple to implement and does not introduce any overhead.

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

A problem with the Round Robin LB is that we do not consider the server load.

As a result, if a server is overloaded or slow, the LB will not stop sending new requests to that server.

To handle this, a more intelligent LB solution can be placed that periodically queries the backend server about its load and adjusts traffic based on that.

10. Purging or DB Cleanup

Should entries stick around forever, or should they eventually be purged?

Is a user-specfied expiration time is reached, what should happen to the link?

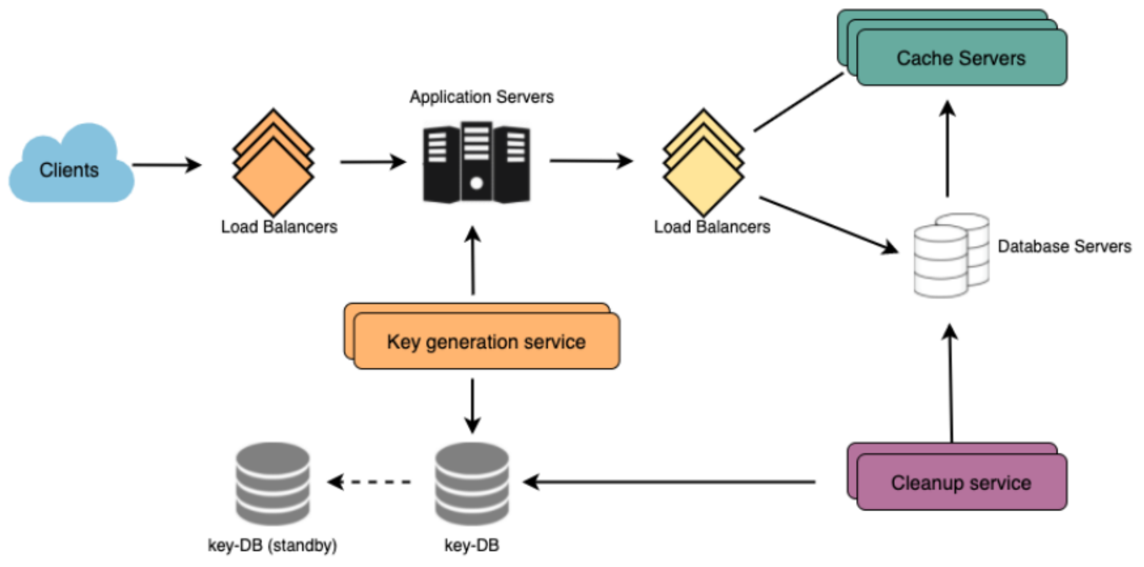
If we chose to continuously search for expired links to remove them, it would put a lot of pressure on our database.

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

Our service will ensure that only expired links will be deleted, although some expired links can live longer but will never be returned to users.

- Whenever a user tries to access an expired link, we can delete the link and return an error to the user
- A separate Cleanup service can run periodically to remove expired links from our storage and cache. This service should be very lightweight and scheduled to run only when the user traffic is expected to be low
- We can have a default expiration time for each link (e.g. 2 years)

- After removing an expired link, we can put the key back in the key-DB to be reused
- Should re remove links that haven't been visited in some length of time, say 6 months? This could be tricky. Since storage is getting cheap, we can also decide to keep links forever



Detailed component design for URL shortening

11. Telemetry

How many times has a short URL been used?

What were the user locations?

How would we store these statistics?

If it is part of a DB row that gets updated on each view, what will happen when a popular URL is slammed with a larger number of concurrent requests?

Some statistics worth tracking:

- Country of the visitor
- Date and time of access
- Web page that referred the click

- Browser or platform from where the page was accessed

12. Security and Permissions

Can users create private URLs or allow a particular set of users to access a URL?

We can store the permission level (public or private) with each URL in the database.

We can also create a separate table to store user IDs that have permission to see a specific URL.

If a user does not have permission and tries to access a URL, we can send an error (HTTP 404) back.

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 user IDs of those users that have permission to see the URL.