**Designing a service like TinyURL –** Reattempt after researching more System Design like approach.

This reattempt was made possible by educative.io’s collection on System Design.

This question was asked by hackernoon.com

**Why do we need a service like TinyURL, how is it important?**

A service such as TinyURL helps reduce large scaled URLs into small scale encrypted URLs that are able to be decrypted and bring you to the base url.

**Requirements/Goals**

Functional Requirements

* URLs must be shorter than what was placed in.
* URLs must be able to redirect the user back to the original url.
* Links have an expiration date (As specified by the final question/statement).
* URLs are able to be customized by selecting from a list of available options.

Non Functional Requirements

* System must be able to handle high demand requests with low latency.
* URLs cannot be predictable.

Other Requirements

* Click count analytics per URL.
* **API support.** - Not based on the question but on educative.io which I agree.

**Estimations, Capacity, Constraints**

Extensive amounts of reads will be occurring. These will be in the form of database lookups and analytical information, all while write will be on the lower end of the spectrum. We can expected a relation of 100:1.

39

**Traffic of the system:** Suppose our system is faced with 100 million URL shortenings a month (1.2 billion a year). On the other side, we would face a 10 billion URL redirects a month (120 billion a year).

Queries per Second (QPS)? → 100,000,000 / (1 month \* 30 days \* 24 hours \* 60 minutes \* 60 seconds) ~ 39 QPS.

Redirections per Second (RPS)? → 39 URL/s \* factor 100 = 3,900 RPS.

**Storage estimates:** From the above estimate of traffic facing 1.2 billion shortenings a year with 120 billion redirects a year. Suppose we run our website for 5 years.

Estimate: Each record held within the system averages to around 500 bytes.

We would only need to calculate the shortenings as redirects don’t consume any additional space.

Storage: 1.2 billion \* 5 years = 6 billion records for a 5 year period.

6 billion \* 500B = **6TB** of additional space required.

**Bandwidth estimates:**

Our write requests were around 39 QPS which when faced with 500 bytes per record, we get 19.5KB/s.

Our read requests are significantly higher but the ratio was 100:1, we can see that 3,900 RPS \* 500 bytes = ~1.95MB/s.

**Memory estimates:** This would be a large portion of interest within our system for the server side of things.

Following the 80-20 rule, 20% of url generates 80% of traffic.

3.9K RPS \* 3600 seconds \* 24 hours = 336.96 million requests per day.

To cache 20% of that would take

336960000 \* .2 \* 500 bytes = ~34GB though due to duplicates, it would be less than this.

Overall estimates

|  |  |
| --- | --- |
| New URLs generated | 39 QPS |
| URL redirects | 3,900 RPS |
| Incoming data | 19.5KB/s |
| Outgoing data | 1.95MB/s |
| Storage for 5 years | 6TB |
| Memory for cache | 34GB |

**System API**

Since we wish to have our System API adhere to REST API standards.

create\_URL(dev\_api\_key, base\_url, custom\_alias=None, user\_name=None, expire\_date=None)

delete\_URL(dev\_api\_key, url\_key)

**Extra:** To prevent abuse, users key will be redirected to x amount of create and deletes within a given time frame. Ex: 50 per month.

**Database Design**

* Viewing our table, we need to store billions of records, around 6TB worth if any haven’t expired.
* Each record is less than 1KB, the record was projected at 500B.
* There are no relation between records as they are completely independent.

**Table – URL**

PK Hash

- Original URL : varchar(256)

- Creation Date : datetime

- Expiration Date : datetime

- UserID : int

**Table – User**

PK UserID

- Name : varchar(20)

- Email : varchar(128)

- CreationDate : datetime

- LastLogin: datetime

**Basic System Design and Algorithm**

**Encoding of the URL – MD5 to B64**

A proposed solution would be to take the base URL provided and convert it into a MD5 hash consisting of 128-bit values or 32-bit hex values. To further process the hash into a *readable* format, a base64 implementation could take place and to take either 6, 8, or even 10 characters from the generated string.

To reconcile the choice, 64^6 =~68.7 billion strings while 64^8 = ~281 trillion possible strings. Depending on our schema, to keep it relatively short, lets stay with 6 characters.

By running through the proposed solution we can see that a base64 generated string is around 21 or more characters. By only selecting the 6 characters, some issues might occur.

* Key duplication may arise in that our selected key has already been generated.
  + Suggested Proposal: Extend the character set for that specific key based on increasing the sequence.

**Offline generation of URL**

Another solution would be to generate all possible 6 character strings beforehand and store them inside our database. Within each new url shortening, a selected string would be assigned from the table to the url. We would have to address that this separate server would need to synchronize with the other servers handing out these keys as issues might arise.

**Concurrency problem**

Using this method, if two of the same url or even two different url were used and the same key was chosen, what would we do. Holding a critical section and having a complete split on two different tables. By dividing up servers and having separate set of keys on each server. Hence users on different servers don’t utilize the same key.

**DB-key size**

6 characters proposed \* 68.7 billion unique keys = 412GB.

**Fall back for KGS**

Since this server is a standalone from the rest, we would need a backup to take over it’s place in the event of failure.

**Data management**

As proposed before, using a **Consistent Hashing** technique would solve any issues on our storage. This storage is based on our generated hash that was found using MD5.

**Cache – Additional for Data management though not in the template**

The required number found for our cache was to view 20% of the total daily traffic consuming the vast majority of our database. We found the 34GB was sufficient space calculated for the system to run, therefore a modern 64GB board would be fine.

To remove dead entries and replace them for newer ones when the cache is full, a LRU algorithm for a Linked Hash Map datastructure would be used. This would allow us to get the LRU entry within O(1) time.

If we fail to get an entry from the cache then we would simply generate a new entry or withdraw an existing since it failed and hit the backend. Distribute this new cache to servers, if a server contains our record, ignore it.

**Networking**

A load balancer would be perfect for this implementation as having a balanced connection between the Client and Application, the Application and Database, and Application and Cache. These load balancers will help differ and balance requests happening, we could even use a round robin to decide what would be the most effective server to use at that time.

**Data cleanup**

To keep track of entries that are expired, we could implement a separate process to scan the database to remove dead entries but that would consume a lot of computing power, placing high strain on the DB. Instead, a few set of rules could also be implemented to ensure that the strain is lessened on the Db.

1. Whenever a user accesses a dead link, remove it from the db.

2. A default expiration date could be setup so the database would be able to track this record.

**Data collection**

We could keep track of geographical information along with ip addresses of places addressed.

**Security**

Due to users being able to generate these URL’s, a token needs to be in place to prevent abuse of generating too many URLs. Though we proposed a dev key limit, a limit on the actual site for an IP using a type of cookie could be used. This cookie would keep track of the computers HWV.