URL shorterner

Wednesday, 6 March 2024 1:39 PM

Lets design a URL shortner, similar to services like Bitly, Tiny URL.
A URL shortener service creates an alias or a short URL for a long URL. Users are redirected to the original URL when they visit these short links.
Need — saves space in general when we share URLs.
— users are less likely to mistype shorter URLs.

Requirements

Functional — Given a URL, our service should generate a shorter and unique alias for it.
— Users should be redirected to the original URL when they visit the short link.
— Link should expire after a default timespan.
Non-functional — high availability with minimal latency.
— system should be scalable and efficient.
Extended — prevent abuse of service.
— record analytics and metrics for redirections.

Estimation and Constraints

Traffic — read heavy system, lets assume 100:1 read/write ratio with 100 million links generated per month.
for reads per month — 100 * 100 million = 10 billion/month
for writes — 1 * 100 million = 100 million/month
— 100 million requests per month translate into 40 requests per second.
$\frac{100 \text{ million}}{(30 \text{ days} * 24 \text{ hrs} * 3600 \text{ seconds})} \approx$ 40 URLs/seconds
with 100:1 read/write ratio, the no. of redirections will be:
100 * 40 URLs/second = 4000 requests/second

Bandwidth — we expect about 40 URLs every second, if we assume each request is of 500 bytes then total incoming data for write requests would be:
40 * 500 bytes = 20 KB/second
— similarly for read requests, since we expect 4K redirections, total outgoing data would be:
4000 URLs/second * 500 bytes $\approx$ 2MB/second

Storage — for storage, we assume we store each link or record in our db for 10 years. Since we expect around 100M requests every month, total no. of records we need to store would be:
100 million * 10 years * 12 months = 12 billion
— assume each stored record will be appox 500 bytes, we need around 6TB of storage.
12 billion * 500 bytes = 6TB

Cache — for cache, we will follow classic Pareto Principle also known as 80/20 rule.
— this means that 80% of requests are for 20% of the data, so we can cache around 20% of our requests.
— since we get 4K read or redirection per second, this translates into 350M requests per day.
4000 URLs/second * 24 hrs * 3600 second $\approx$ 350 Million requests/day
— Hence we need 35GB of memory per day
20 percent * 350 million * 500 bytes = 35GB/day

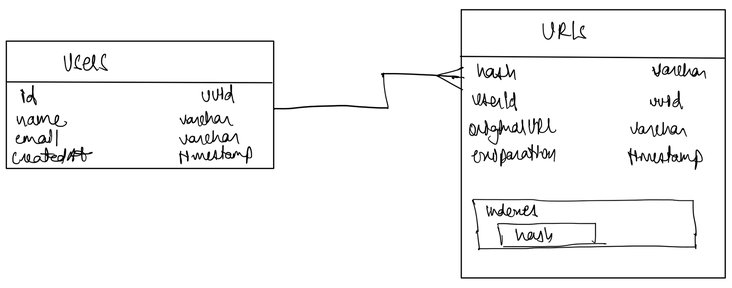
High-level estimate

Type	Estimate
writes (new URLs)	40/s
Reads (redirection)	4K/s
Bandwidth (incoming)	20 KB/s
Bandwidth (outgoing)	2MB/s
Storage (10 years)	6TB
Memory (caching)	~35 GB/day

Data model Design

users — stores user details such as name, email, created At etc.
URLs — contains new short URL's properties such as expiration, hash, original URL and user ID of who created the short URL.
— we can also use hash column as an index to improve the query performance.

Database kind — since data is not strongly relational, NoSQL db such as Amazon DynamoDB, Apache Cassandra or MongoDB will be a better choice here.



API Design

Create URL — should create a new short URL in our system given an original URL.
createURL (apiKey: string, originalURL: string, expiration?: Date): string
Get URL — should retrieve original URL from a given short URL.
getURL (apiKey: string, shortURL: string): string
Delete URL — should delete a given shortURL from our system.
deleteURL (apiKey: string, shortURL: string): boolean

Need of API key — we are using an API key to prevent abuse of our services. Using this API we can limit the users to a certain no. of requests per second or minute.
— This is quite a standard practice for developer APIs and should cover our extended requirements.

High Level Design

URL encoding — primary goal is to shorten a given URL, different approaches:

Base62 Approach

— we can encode original URL using Base62 which consists of A–Z, a–z and 0–9.
no. of URLs = 62^N where n is no. of characters in generated URL
— simplest solution but does not guarantee non-duplicate or collision-resistant keys.

MD5 Approach

— MD5 message-digest algorithm is widely used hash function producing a 128-bit hash value (or 32 hexadecimal digits).
— we can use these 32 hexadecimal digits for generating 7 characters long URL.
MD5(original URL) ⟶ base62 encode ⟶ hash
— However, this creates new issue, which is duplication and collisions.
— we can try to re-compute the hash until we find a unique one but that will increase the overhead of systems (better to look for more scalable approach)

Counter Approach

— In this approach, we will start with a single server which will maintain the count of keys generated.
— once our service receives a request, it can reach out to the counter which returns a unique number and increments the counter.
— when next request comes the counter again returns the unique no. and this goes on.
counter (0–3.5 trillion) ⟶ base62 encode ⟶ hash

— problem with this approach is that it can become a single point of failure. And if we run multiple instances of the counter we can have collisions as it's essentially a distributed system.
— To solve this, we can use a distributed system manager such as Zookeeper which can provide distributed synchronization. Zookeeper can maintain multiple ranges for our servers.
Range 1 : 1 ⟶ 1,000,000
Range 2 : 1,000,001 ⟶ 2,000,000
Range 3 : 2,000,001 ⟶ 3,000,000
— Once a server reaches its maximum range Zookeeper will assign an unused counter range to new server. This approach can guarantee non-duplicate and collision-resistant URLs.
— Also, we can run multiple instances of Zookeeper to remove single point of failure.

Key Generation Service (KGS)

— Generating a unique key at scale without duplication and collisions can be a bit of challenge.
— To solve this we create a standalone KGS that generates a unique key ahead of time and stores it in a seperate database for later use.

How to handle concurrent access?
— Once key is used, we can mark it in the db to make sure we don't reuse it, however, if there are multiple server instances reading data concurrently, two or more servers might try to use the same key.
— easiest way to solve this would be to store keys in two tables. As soon as a key is used, we move it to seperate table with appropriate locking in place.
Also to improve reads, we can keep some keys in memory.

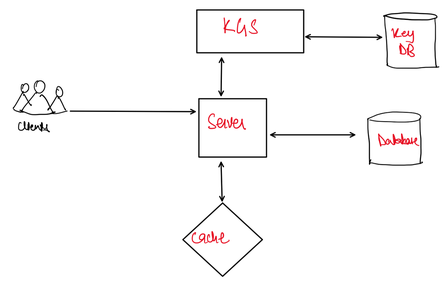
KGS database estimations
— As per discussion, we can generate up to ~56.8 billion unique 6 character long keys which will result in us having to store 300GB of keys.
6 characters * 56.8 billion $\approx$ 390GB
— while 390 GB seems like a lot for this simple use case, it is important to remember this is for the entirety of our service lifetime and the size of the keys db would not increase like our main database.

Caching

— as per estimate, we require around ~35GB of memory per day to cache 20% of incoming requests to our services.
— we use redis or memcached servers alongside our API servers.

Design

— Now we have identified core components, lets do first draft of system design.



Working
⟶ Creating a new URL
a) when user creates a new URL, our API server requests a new unique key from the KGS.
b) KGS provides a unique key to API server and marks the key as used.
c) API server writes the new URL entry to the database and cache.
d) Our service returns an HTTP 201 (Created) response to user.
⟶ Accessing a URL
a) when client navigates to a certain short URL, the request is sent to API servers.
b) Request first hits the cache, and if the entry is not found there then it is retrieved from the db and HTTP 301 (redirect) is issued to original URL.
c) If the key is still not found in db, an HTTP 404 (not found) error is sent to the user.

Detailed Design (discuss finer details of our design)

Data Partitioning

— to scale out our db we will need to partition our data. (Sharding can be a good step)
— we can use partition schemes such as — hash-based partitioning
— list-based "
— range-based "
— composite "
— Above approaches can still cause uneven data and load distribution, we can solve using consistent hashing.

Database Cleanup

— this is more of a maintenance step for our services and depends on whether we keep the expired entries or remove them.
If we do decide to remove expired entries we can approach in two different ways:
— Active CleanUp
We will run a seperate cleanUp service which will periodically remove expired links from our storage and cache. This will be a very lightweight service like a cron job.
— Passive CleanUp
For passive cleanup, we can remove the entry when a user tries to access an expired link. This can ensure a lazy cleanup of our db and cache.

Cache

— LRU can be a good policy for our system. We discard the least recently used key first.
— How to handle cache miss?
— whenever there is a cache miss, our servers can hit the db directly and update the cache with the new entries.

Metrics and Analytics

— Recording analytics and metrics is one of our extended requirements. We can store and update metadata like visitor's country, platform, the no. of views etc alongside URL entry in our db.

Security

— for security, we can introduce private URLs and authorization.
— a seperate table can be used to store user ids that have permission to access a specific URL. If user does not have a proper permissions, we can return an HTTP 401 (unauthorized) error.
— we can also use an API Gateway, as they can support capabilities like authorization, rate limiting and load balancing out of the box.

Identify and Resolve Bottlenecks

— lets identify bottlenecks such as single point of failure in our design:
— what if API service or key generation service crashes?
— How will we distribute our traffic between our components?
— How can we reduce the load on our database?
— what if the key db used by KGS fails?
— How to improve the availability of our cache?

— To make system more resilient we can do:
— Running multiple instances of our servers and KGS.
— Introducing load balancers between clients, servers, db and cache servers.
— Using multiple read replicas for our db as the a read-heavy system.
— Standby replica for our key db in case it fails.
— Multiple instances and replicas for our distributed cache.