URL Shortening Service

It is inconvenient to share a huge URL. Therefore, some internet platforms such as tinyurl.com offer services to generate a short URL from a longer URL.

1. Requirements

The requirements of the system are as follows:

- Functional Requirements:
  - When a long URL is provided, a short link is generated.
  - Should generate a unique link.
  - Needs to redirect to the correct URL.
  - There needs to be an expiration date for the short URL.
  - A provision for a custom alias.

- Non Functional Requirements:
  - Availability over Consistency
  - Need low latency, Redirect in real time.
  - Reliability
  - Security

2. Capacity Estimation

We shall consider the following parameters for capacity estimation:
- Traffic (Read/Write Requests as Queries per second)
- Storage
- Bandwidth
- Memory

1. Traffic

Assume 100 million users per month with each user creating an average of 2 URLs per month.

Therefore, we have 100M * 2 URLs/ (30 * 24 * 60 * 60) = approximately 80 write queries per second generated.

Our system is assumed to be read heavy in the ratio 80:1.

Read is to basically access a URL while write is to generate a URL.

Therefore, traffic is 80 * 80 = 6400 Read Queries per second and 80 Write Queries per second for 100M users every month.

So approximately 8 Billion Reads/month assuming 80 reads for one write * 2 URLs/user. So that gives us 16 Billion Reads/month for 200M writes/month, thus the ratio 80:1.

2. Storage (Disk Space)

100M * 2 URLs/user

Assume URL Metadata is 0.25kB per User or 250 Bytes. We shall assume expiration time as 3 years.

So we have:

- 100M users/month
- 2 URLs/month per user
- 250 Bytes per URL
- 3 years Expiration Date
- 12 months/year

So total storage: = 100M * 2 * 250 Bytes * 3 * 12 = 25 * 10^8 * 72 = 1.8TB

3. Bandwidth

- Incoming Data (Writes)
  - 80 * 250 Bytes = 20 kbps
- Outgoing Data (Reads)
  - 6400 * 250 Bytes = 1.6 mbps

4. Memory (Cache)

We apply the 80:20 Principle where we assume 20% of the URLs make up 80% of the traffic. Therefore, we cache 20% of the requests.

There are 6400 read queries/sec. Therefore, per day there are 6400 * 3600 * 24 = Approximately 550M requests per day.

We assume cache refresh every day. We cache 20% of these requests. Therefore, we have 550M * 0.2 * 250 Bytes = 27.5 GB per day cache memory.

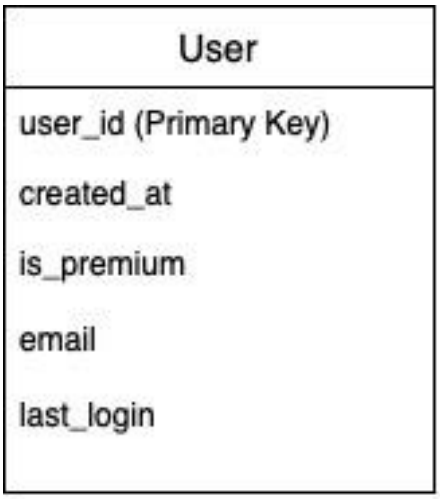
Let us keep a buffer as 50 GB for peak days.

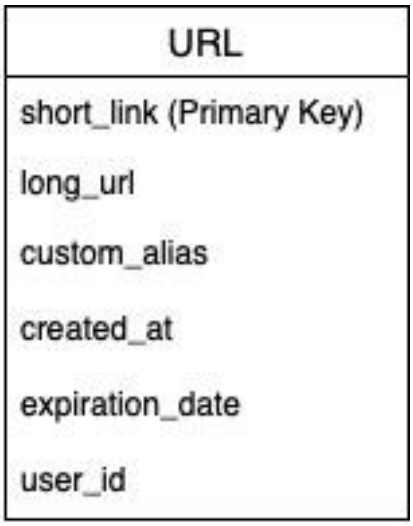
3. System APIs

- createURL(api_dev_key, url_long, url_id, custom_alias, created_at)
- deleteURL(api_dev_key, short_link)

We will assign an api_dev_key uniquely to a user for checking system inundation with request bombing by any user.
There will be a limit on the length of the custom Alias. Also, the system should not let a user delete another user's URL.

4. Data Modeling





Criteria:

- Billions of requests
- Table rows not necessarily related
- Availability over Consistency
- Read Heavy

Relational Databases are more aligned towards consistency than availability. Hence, we shall use a NoSQL database. NoSQL scales well for availability.

We need an AP Database (based on CAP Theorem) as we give more focus to availability than consistency. Cassandra and DynamoDB are the options we have for AP Databases.

5. High Level Design

MD5 generates a 128 bit hash although our URL is 6 – 8 bits. Therefore, we use base64 encoding. Base64 converts the 128 bit hash provided by MD5 to 21 bits after dividing by 6. However, we require only 6 bits. So we generate a link till it is unique by comparing it against the database, by using hashing plus encoding.

We could also use a key generation service to avoid the hassle of hashing and encoding. A key generation services generates a random unique key through a custom logic which can be used as the unique ID for the short URL.

One approach to manage key generation collisions is to use a used v unused table where keys already generated are stored in the used table after they are assigned. However, this may cause latency issues as keys will be compared and fetched at each query. Hence, an optimized approach is to use a cache of pre-generated keys from the KGS and provide the cached keys to the app server. As soon as a short link is allocated, it is removed from the cache and stored in the used keys table.

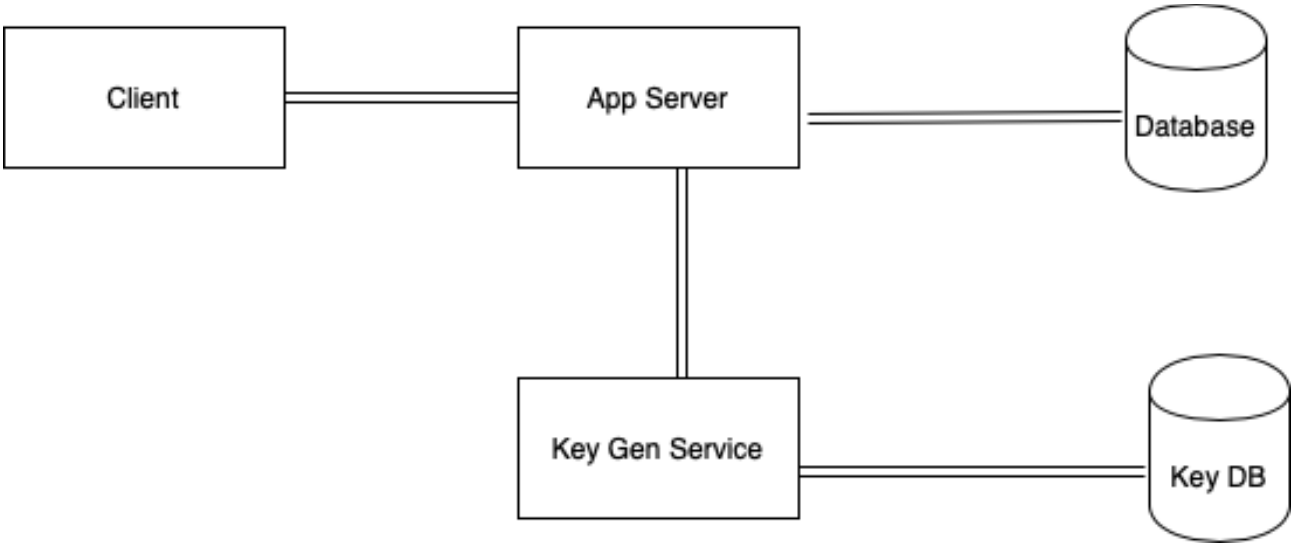


Figure: KGS High Level

6. Component Design

1. Partitioning

We use partitioning to divide the database into n partitions to ease the load on a particular database. Distributing data into different databases is called range based partitioning. However, range based partitioning is imbalanced i.e. some partitions are underutilized. Another technique is hash based partitioning. Here we calculate a hash to determine which partition to store in. Because of the randomness, there is a better balance.

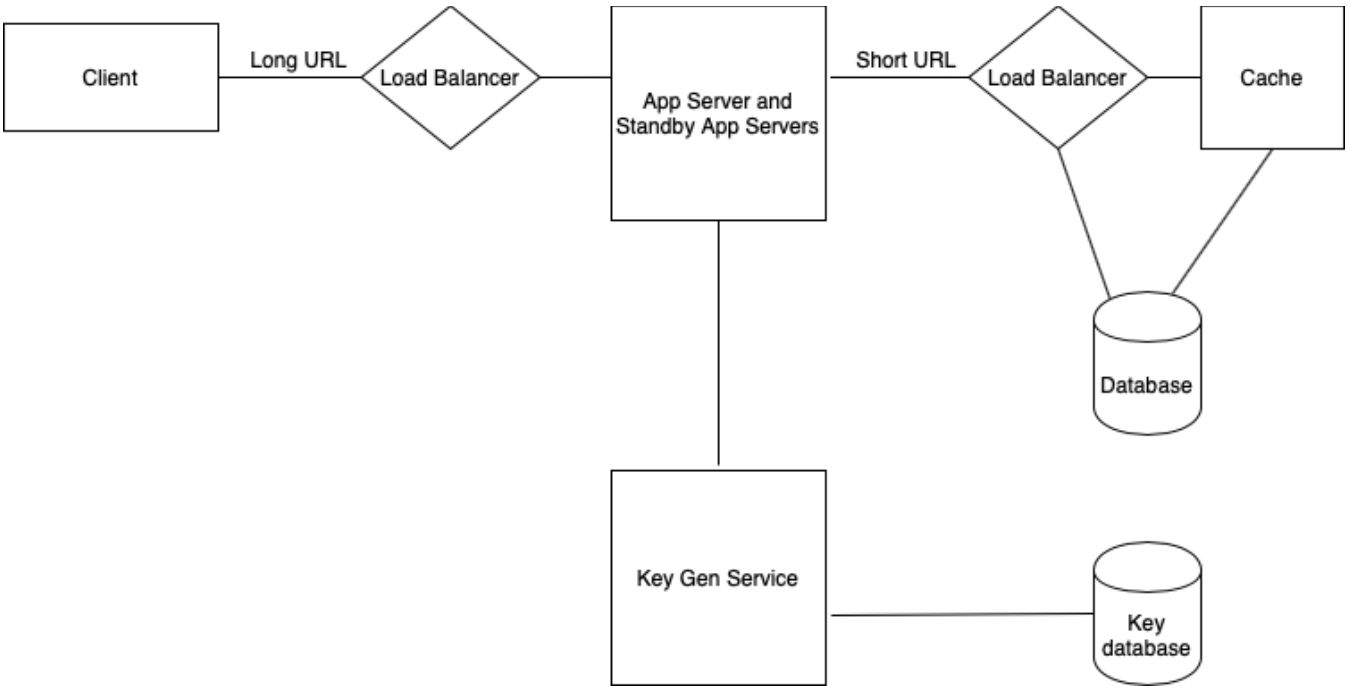
2. Caching

You would create a URL only when required and for immediate consumption. Hence, you would require the most recently used value. Therefore, the caching technique to use is the Least Recently Used (LRU) cache. Further, we would use a low latency write back technique i.e. update DB sometime after cache is updated and fetch immediately from cache instead of DB.

3. Load Balancing

The load balancing technique to be used here is the least response time or weighted round robin. We should use least response time when all the servers are equal size, otherwise we use weighted round robin for imbalanced server capacities.

7. Low Level Design



8. Clean up

URLs that have expired need to be cleaned up. We shall run a CRON job as scripts could burden the system. We should schedule a CRON job to run once a week at times of least system usage.

Another choice is to use lazy cleanup every 6 months and if user accesses an expired URL, calculate the difference from timestamp and delete it from the database. Also, we could move deleted keys to the key DB to reuse the keys for another URL.

9. Telemetry (Monitoring and Analytics)

We could analyze patterns of traffic to understand which URLs are accessed more often. Currently we have an average of 6400 read queries per second. We could record and partition the URLs which make up a significant portion of these read queries. Say a URL is accessed for 10% of the read queries i.e. 640 times, against other URLs accessed an average of 1% that is only 64 times, we could save it as a hot URL. This could help us in caching and regenerating short URLs for popular URLs and/or web pages which could improve system performance.

We could also use the api_dev_key to highlight users who create the highest number of URLs to identify potential promotional offers for these users. This would help increase revenue from the system.

We could also use the geolocation and IP address from where a request was made to understand topics and web pages of interest in a particular region and also identify places which generate maximum traffic for the URL shortening site.