**Understanding the problem**

Designing a URL Shortening Service. The goal is to design a highly scalable service that could allow users to create shorter URLs, given long URLs, and have read and write functionality.

**Requirements of the System**

The URL Shortening service should have the following features:

**Functional Requirements**

* Users should be able to generate a shortened URL from the original URL.
* The short link should redirect users to the original link.
* Users should have the option to give a custom short link for their URL.

**Non-Functional Requirements**

* If the system fails, it will imply that all the short links will not function. Therefore, our system should be highly available.
* URL redirection should happen in real-time with minimal latency.
* Shortened links should not be predictable in any manner.

**System Analysis**

Now we will look at the estimate of number of monthly requests, storage, etc. We are given the read/write = 100:1.

**Key components**

Following will be the key components in the URL shortening application:

1. **Clients** – Web Browsers/Mobile app. It will communicate with the backend servers via HTTP protocol.
2. **Load Balancer** – To distribute the load evenly among the backend servers.
3. **Web Servers** – Multiple instances of web servers will be deployed for horizontal scaling.
4. **Database** – It will be used to store the mapping of long URLs to short URLs.

**Estimation of Scale and Constraints**

**Traffic Estimation**: We are given that there will be having 5M new URL shortening requests per month, with a 100:1 read/write ratio, so we can expect 500M redirections during the same period(100\*5M=500M).

* If we calculate this value for each second, i.e., queries per second (QPS) = 5M/ (30 days\*24 hours\*3600 seconds) = ~2URLs/s.
* Considering the 100:1 read/write ratio, URLs redirections per second will be = 100\*2 URLs/s = 200 URLs/s.

**Storage Estimation**: Let us assume that we are willing to store our data (short URL + long URL) for ten years then, the number of URLs we will be storing would be = 5M\*10\*12 = 600M URLs.

Now, we assume the URL’s length to be 120 characters (120 bytes) on average and then add 80 bytes to store the information about the URL. Total storage requirement = 600M\*200 bytes = 120 GB.

**Encoding URL**

We will use (a-z, A-Z, 0-9) to encode our URLs which leads us to use base62 encoding. The longer our key, the more URLs we have, and less we have to worry about our system ever running out. However, the point of this system is to keep our URL as short as possible. Let’s stick with **7**.

**Encoding**

To generate a unique short URL, we can compute it using the **Unique Hash (MD5, SHA256, etc.)** of the original URL and then encode using **base62**. If we use the MD5 algorithm as our hash function, it’ll produce a 128-bit hash value. After base62 encoding, we’ll get a string having more than seven characters, we can take the first **7 characters** for the key.

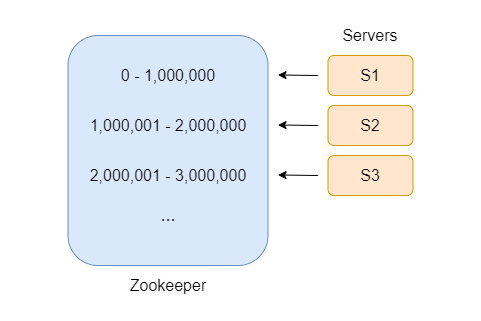
There is a problem with this approach. This could (however unlikely) result in **duplication or collisions** in the database. One of the **solutions** to this problem is to use a **counter**, which keeps track of a count (0-3.5 Trillion) so that in each case if we encode the counter value, there is a guarantee of no duplicates/collisions. We will use a single server that will be responsible for maintaining a counter. Whenever the application servers receive a request, they will talk to the counter, which returns a unique number and increments the counter.

**Two potential problems**

1. Single Point of failure.
2. If requests spike, our counter host may not be able to handle it.

In this case, we will use a **distributed systems manager**, such as **Zookeeper**. Let us see how a distributed systems manager like Zookeeper solves our problem:

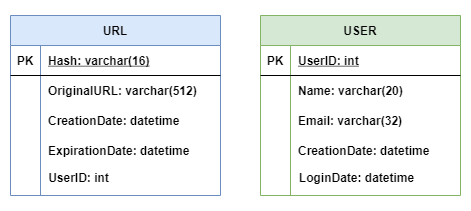
1. It gives our application servers unused ranges.
2. To avoid having Zookeeper be a single point of failure, we will run **multiple instances**.
3. If a new server is added, it will give them an unused range. If the range runs out, the existing server can go to Zookeeper and ask for a new, unused range.
4. If one of the **servers dies**, **1 million keys** are possibly **wasted**, which is **acceptable**, given the 3.5 trillion keys we have.
5. However, sequentially generating URLs can be a security threat. We can add another random 10-15 bits after the counter number to increase the randomness.



**Database design**

While designing systems, we have two types of databases (of course, we don’t want to back to the old days where file systems were used): **Relational Databases** or **NoSQL**. For our system, relational queries will not be a large part of it/occur rarely. So, here we will go with NoSQL. **A NoSQL choice would be easier to scale**.

**Database schema**: We will need two main tables: 1 to store user information and 1 to store URL information.



**Scaling the service**

**Caching**

We know that our database is going to be read heavily. We have found a way to speed up the writing process, but the reads are still slow. So, we have to find some way to speed up the reading process. Let’s see.

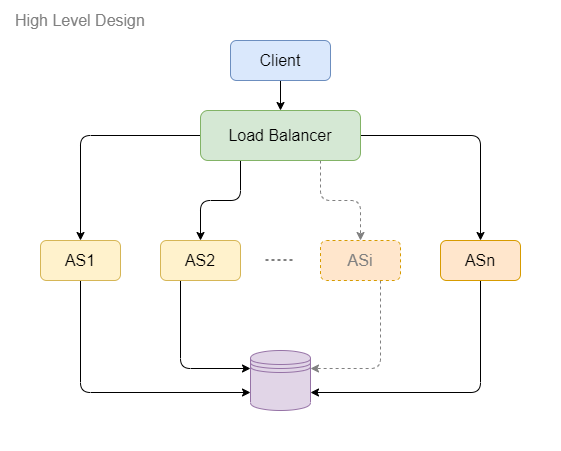
We can speed up our database reads by putting as much data in memory as possible, AKA a **cache**. This becomes important if we get massive load requests to a single link. If we have the redirect URL in memory, we can process those requests quickly. Our cache could store, let’s say, **20% of the most used URLs**. When the cache is full, we would want to replace a URL with a more popular one.

A **Least Recently Used (LRU)** cache eviction system would be a good choice. We can **shard** the cache too. This will help us store more data in memory because of the more machines. Deciding which thing goes to which shard can be done using “**Hashing**” or “**Consistent Hashing**”.

**Load Balancing**

With multiple access systems like this one, a web server may not handle it yet. To solve this problem, I will use many web servers. Each web server will take a partial request from the user.

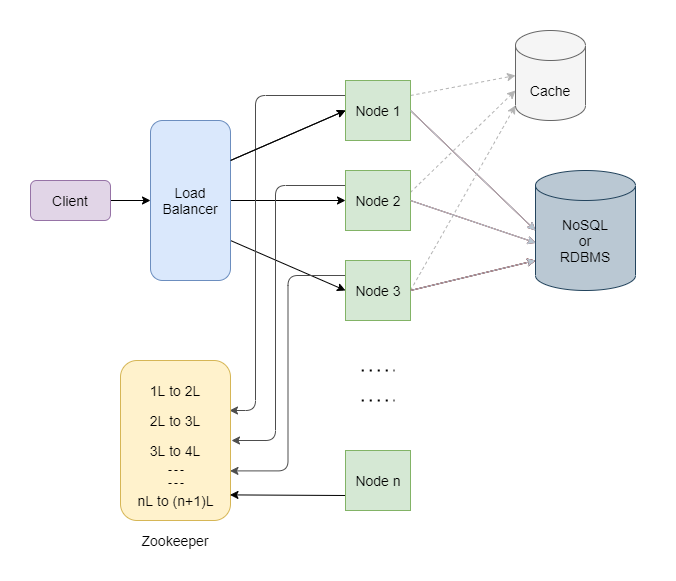
Since we have a large-scale system with multiple access happening simultaneously, a server may not handle the requests. To solve this problem, we can use multiple servers with a load balancer between “**the client and the server**” and “**the server and the database**” and avoid a single point of failure, and we will use multiple load balancers.



Initially, we can use a simple **Round Robin** approach that distributes incoming requests equally among backend servers. This would be the easiest to implement. A Round Robin LB does not consider server load, so our LB may still forward requests to an overloaded or slow server.

To distribute the load among servers, we will use the **Least Connection Method.** When a server is configured to use the least connection load balancing algorithm (or method), it selects the service with the fewest active connections.

**Skeleton of the design**



Hence in this way, we could design a highly scalable **URL shortening service**.