SCALABLE, RELIABLE, EFFECIENT, ZERO COLISSIONS

The ones listed above are the main features of the app which is essential for a URL shortener service. Every component is designed to adhere to the above features. The design choice and reasoning for the same is listed below:

DB Design:

The two choices were RDBMS and NoSQL. NoSQL was chosen as there is not a lot of relationships in our data

and NoSQL is faster for writing and simple key-value reads.

The MAIN advantage of NoSQL is its ease to scale by adding more servers to a distributed system hence I have used MongoDB Atlas it would also overcome single point of failure.

I have used 3 schemas defined below:

url schema: { longUrl, shortUrl, user, creationDate}

user schema: {userID, tier, requests}

tier schema: {1:5000,

2:1000,

3:100,

4:5

}

Short URL generation:

We could generate a random URL and try to insert it to the DB using the putIfAbsent clause, but this may result in a lot of collisions. Additionally, we could use a MD5 or SHA256 algorithm to generate a unique value but as the size of hashed output is more than 7, we would have to use only the first 7 or 8 characters which would again lead to collisions.

Instead, we could maintain a counter, which we would pass to a base62 encoder that would result in a unique value every time we generate a short URL resulting in zero collisions. To overcome a single point of failure and maintain the counter across all servers we use the Apache ZooKeeper which is a highly reliable distributed coordinator.

If we use 7 a short URL of 7 chars using base 62 we have about 3.5 Trillion unique short URLs we could generate.

ZooKeeper maintains ranges of unused counters. When servers are added these servers will ask for the unused range from Zookeepers. In the worst case, if one of the servers goes down then only that range of data is affected. We can replicate data of master to its slave and while we try to bring master back, we can divert read queries to its slaves.