A URL shortening service transforms long, complex URLS into concise, user-friendly links, reducing the length and complexity. It necessitates consideration for unique key generation, data persistence, high availability, and redirect speed.

**REQUIREMENTS**

**FUNCTIONAL**

1. Take a URL and return a much shorter URL.

2. Take a short URL and redirect to original URL.

3. Allows user to pick custom short URL.

4. Usage statistics for site owners lie to know how many people clicked the shortened URL in last day.

**NON-FUNCTIONAL REQUIREMENTS**

1. Consitency The system should ensure data consistent across multiple servers).

2.Availability The system should be highly available)

3.Partition Tolerance the system should be able to function and maintain the data integrity even when network partitions occur) )

4.Scalability the system should be able to handle the increasing number of request 5.Performance short URLS should be generated and resolved quickly)

6. Latency

**ESTIMATIONS:**

Daily Active User(DAU) = 1B( social Media Users)\*.01(percent that post shortener links each day)\*.25(our market share) = 2.5MDAU

Storage Requirements:

1. dbRow = 100b(orignal URL)+8b(short Url)+500b(metadata&analytics) = about1KB

2.5M(dau)\*0.01(100:1 read to write ratio)\*1(dbRow)\*2(redundancy)\*2(backup)\*2(growth)\*365(days peryear)\*10(10yrs)

= around 1TB.

**API Design**

**Shortening URL(URL)** - Creates a new short URL from the given long URL

POST/shorten

REQ: {URL:'long url',userId(optional)}

RES: {shortURL:'generated short URL'}

**RedirectionURL(url)**

GET/:shortURl

Response: redirects to original URL

GET/stats

Response:{'shortUrl':'short URL','URL:'original URL', 'click':number of redirect happend'}

POST/users

req: name, email

Response: id

GET/users/:id

response: id,name,email

**DATABASE SCHEMA**

**User**:

1. id string

2. Name String

3. email string

**Short Url**

1. Id string

2. shortUrl string

3. originalURl: String

4. userId String(reference User.id)

5.timestamp

**Stats**

1. id string

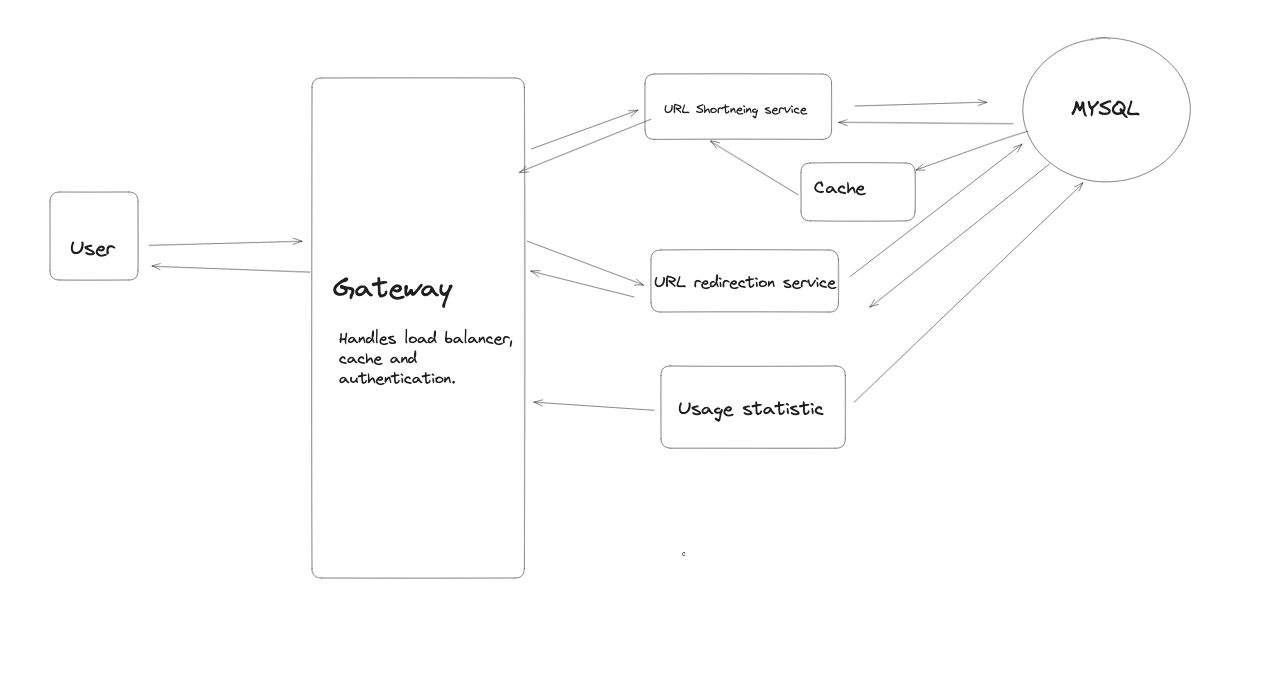
2. shortUrlId string(reference to shortUrlId)

3. clickedAt: datetime

4. ipAddress: string

5. userAgent: string

**HIGH LEVEL DESIGN**



So, I would use a **NoSQL database**, specifically a key-value store like Redis. The main reason for this choice is that Redis excels at simple key-value mapping, which is essential for the URL shortening service. When we prioritize speed, it is crucial for providing a responsive service. From such use cases, performance is the key, which Redis offers with its in-memory data storage."

To generate a **shorter and unique alias for a given URL**, we can use a combination of hashing and encoding techniques. First, we can create a hash of the input URL using a hashing algorithm like MD5 or SHA-1. Then, we can encode the hash using a URL-friendly character set, such as Base62 (which includes alphanumeric characters: A-Z, a-z, and 0-9). Finally, we can truncate the encoded hash to a desired length, like 6 or 8 characters, to create a short alias. As we discussed earlier, this will give us 62^6 = 58 billion unique combinations, which is far more than enough unique combinations to satisfy our usage over the life of the system. Another option would be to use a separate ID generation service to increment a unique ID for each new url. Maintaining a globally unique and incrementing counter is challenging in a distributed system due to issues like synchronization and scalability. This is why I'm deciding to go with the hashing approach instead."

###### **Horizontal & Vertical Scaling**

Horizontal scaling adds more machines, while vertical scaling increases the capacity of an existing machine.

###### **Load Balancing**

Distributes the workload evenly among servers, enhancing efficiency and reliability.

###### **Data Partitioning & Caching**

Data partitioning manages large databases more efficiently. Caching enhances performance by storing frequently accessed data.

**The primary bottleneck** in our URL shortening service could be the latency in the retrieval of the original URL from the shortened one, especially under high loads. This is essentially a read operation on our NoSQL key-value store database. Given the nature of a URL shortening service, which is read-heavy, this operation needs to be as quick as possible for a good user experience. To mitigate this bottleneck, we can employ the following strategies: 1. Caching: A caching layer can be introduced between the application layer and the database. The cache will store frequently accessed short URLs and their corresponding original URLs. This will significantly reduce read operations on the database. A cache eviction policy (like LRU) can be used to manage the cache size. Tools like Redis or Memcached can be effectively used for this purpose. 2. Read Replicas: Even though NoSQL databases are known for their high read and write throughputs, under extreme loads, we can create read replicas of our database. This will distribute the read load across multiple replicas, thus preventing any single database instance from becoming a bottleneck."