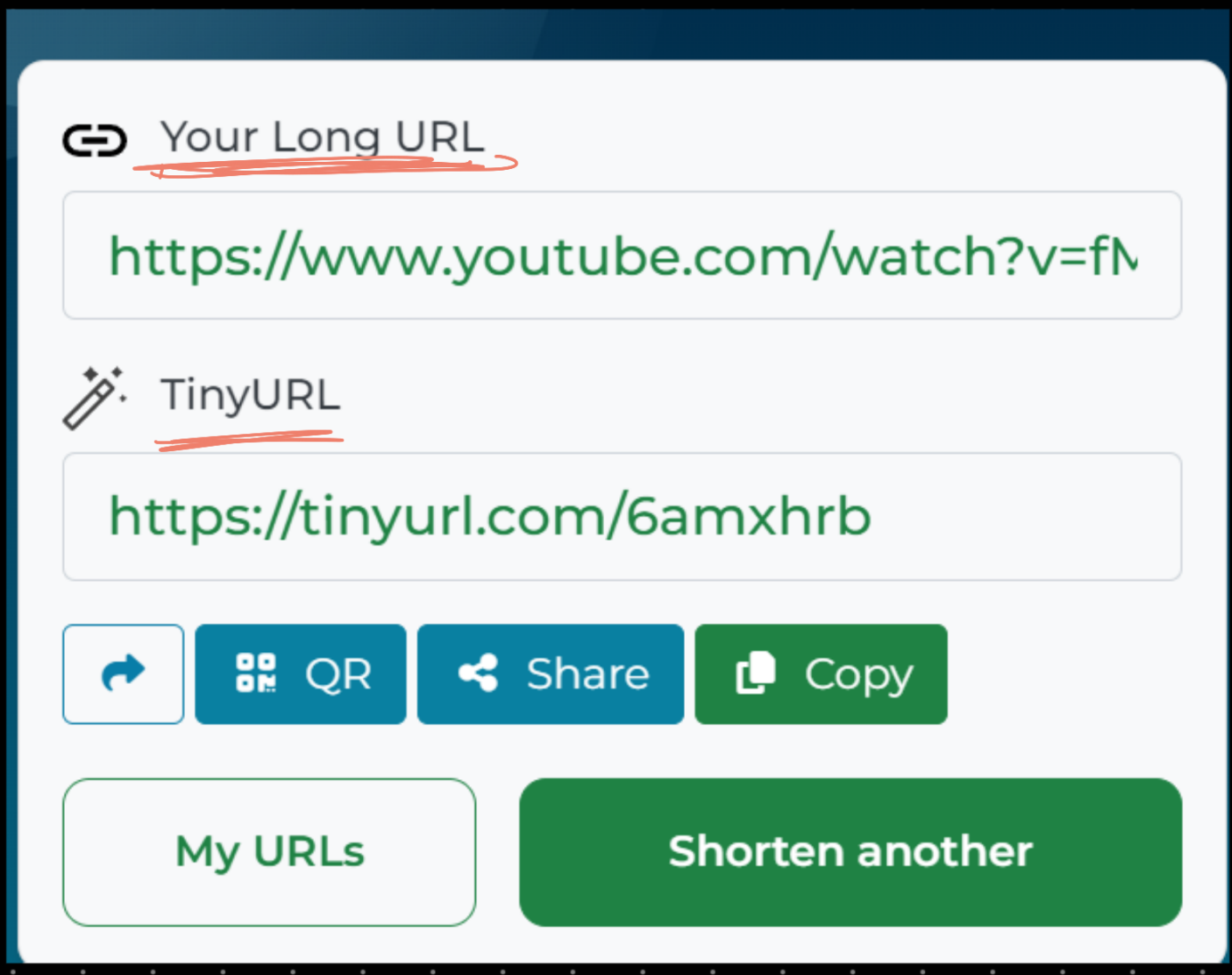
[URL shortner System Design]



Functional Requirements:

- Generate a unique short URL for a given long URL
- Redirect the user to the original URL when the short URL is accessed
- Allow users to customize their short URLs (optional)
- Support link expiration where URLs are no longer accessible after a certain period
- Provide analytics on link usage (optional)

Non-Functional Requirements:

- High availability (the service should be up 99.9% of the time)
- Low latency (url shortening and redirects should happen in milliseconds)
- Scalability (the system should handle millions of requests per day)
- Durability (shortened URLs should work for years)
- Security to prevent malicious use, such as phishing.

2. Capacity Estimation

Let's assume the following traffic characteristics:

> **Daily URL Shortening Requests**: 1 million requests per day
>
> **Read:Write ratio:** 100:1 (for every URL creation, we expect 100 redirects)
>
> **Peak Traffic**: 10x the average load
>
> **URL Lengths**: Average original URL length of 100 characters

2.1 Throughput Requirements:

- **Average Writes Per Second (WPS)**: (1,000,000 requests / 86,400 seconds) ≈ 12
- **Peak WPS:** 12 ×10 = 120

Since Read:Write ratio is 100:1

- **Average Redirects per second (RPS)**: 12 * 100 = 1,200
- **Peak RPS:** 120 * 100 = 12,000

2.2 Storage Estimation:

For each shortened URL, we need to store the following information:

- **Short URL**: 7 characters (Base62 encoded)
- **Original URL**: 100 characters (on average)
- **Creation Date**: 8 bytes (timestamp)
- **Expiration Date**: 8 bytes (timestamp)
- **Click Count**: 4 bytes (integer)

Total storage per URL:

- **Storage per URL**: 7 + 100 + 8 + 8 + 4 = 127 bytes

Storage requirements for one year:

- **Total URLs per Year**: 1,000,000 × 365 = 365,000,000
- **Total Storage per Year**: 365,000,000 × 127 bytes ≈ 46.4 GB

2.3 Bandwidth Estimation:

Assuming the HTTP 301 redirect response size is about 500 bytes (includes headers and the short URL).

- **Total Read Bandwidth per Day**: 100,000,000 × 500 bytes = 50 GB / day
- **Peak Bandwidth**: If peak traffic is 10x average, the peak bandwidth could be as high as 500 bytes × 12,000 RPS = 6 MB/s

2.4 Caching Estimation:

Since it's a read-heavy system, caching can significantly reduce the latency for read requests.

If we want to cache some of the hot URLs, we can follow the **80-20 rule** where 20% of the URLs generate 80% of the read traffic.

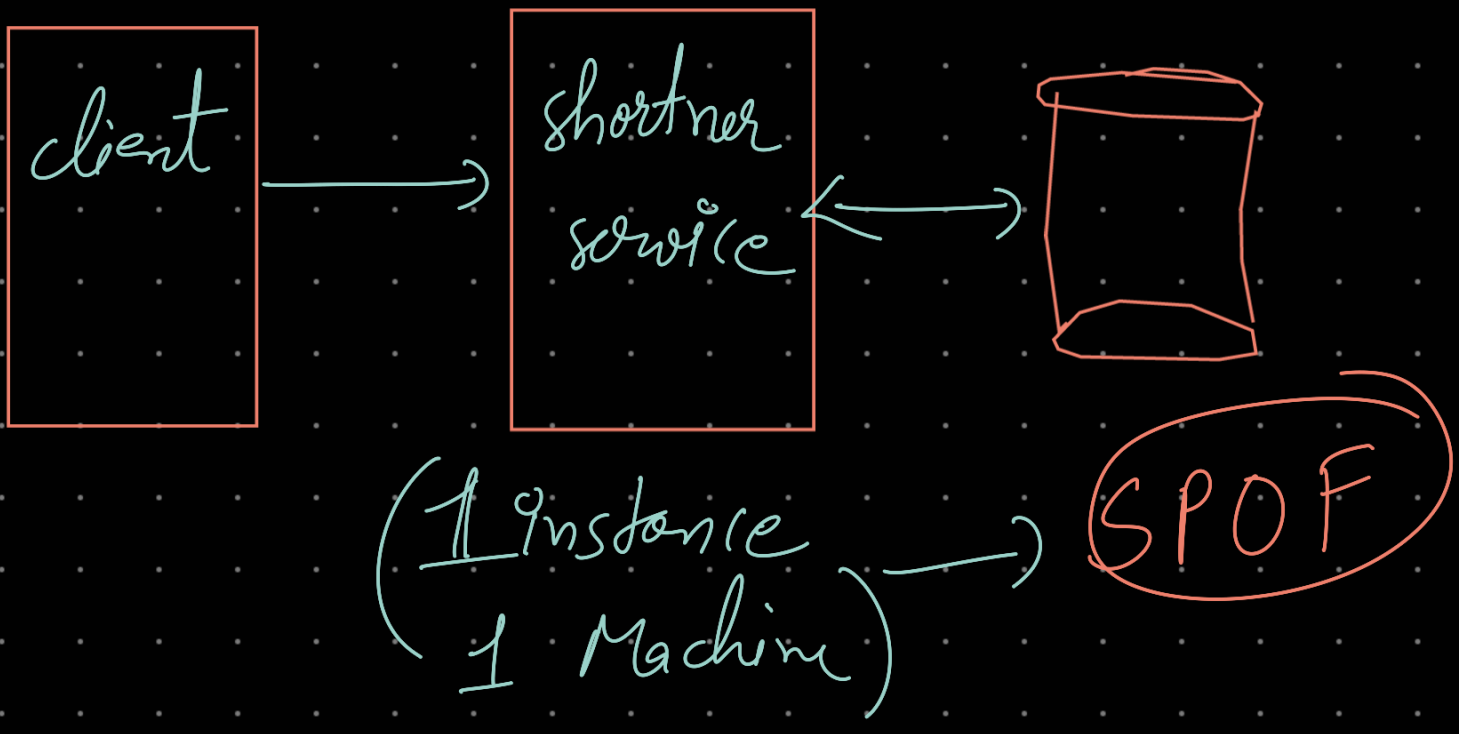
Since we have 1 million writes per day, if we only cache 20% of the hot urls in a day,
Total cache memory required = 1M * 0.2 * 127 Bytes = 25.4 MB

Assuming a cache hit ratio of 90%, we only need to handle 10% of the redirect requests directly from the database.

Requests hitting the DB: 1,200 × 0.10 ≈ 120 RPS

This is well within the capabilities of most distributed databases like DynamoDB or Cassandra, especially with sharding and partitioning.

For small scale, Design is simple



For small scale no problem

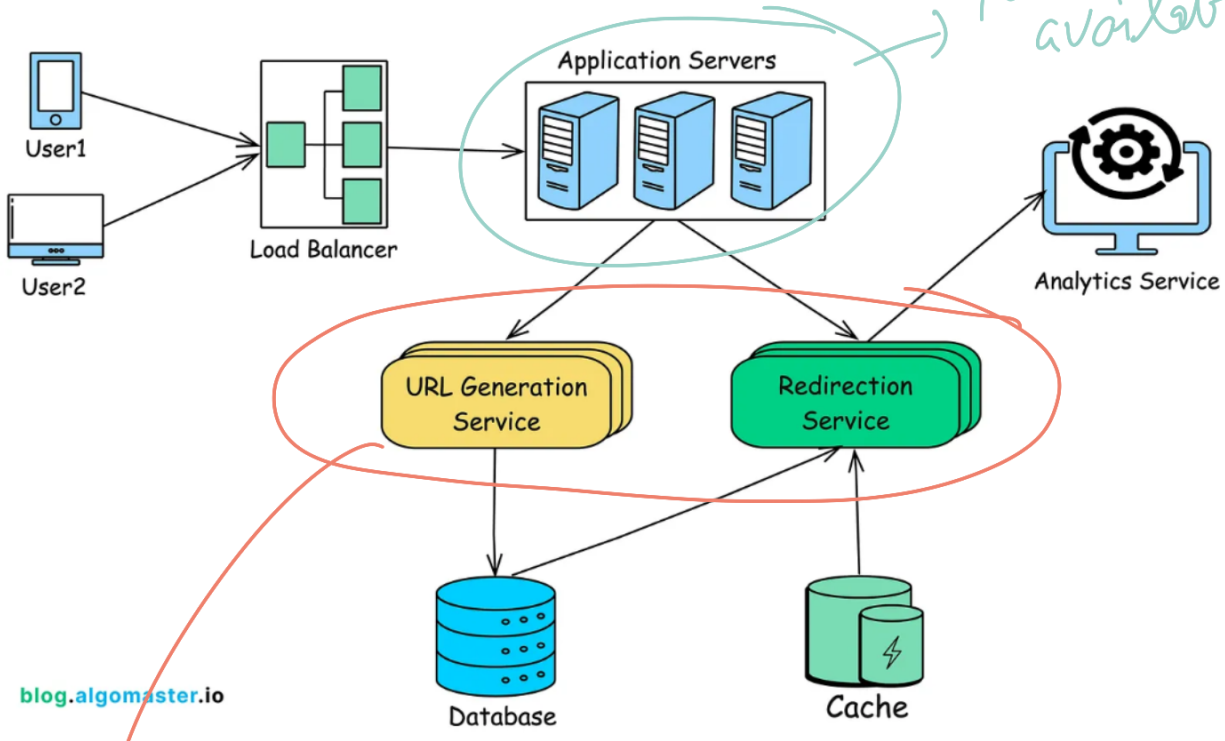
2.5 Infrastructure Sizing:

To handle the above estimations:

- **API Servers**: Start with 4-6 instances behind a load balancer, each capable of handling 200-300 RPS.
- **Database**: A distributed database with 10-20 nodes to handle both storage and high read/write throughput.
- **Cache Layer**: A distributed cache with 3-4 nodes, depending on the load and cache hit ratio.

3. High Level Design

On a high level, we would need following components in our design:



For High availability

we can have monolith for both service.
↳ No need of Microservice

4. Database Design

4.1 SQL vs NoSQL

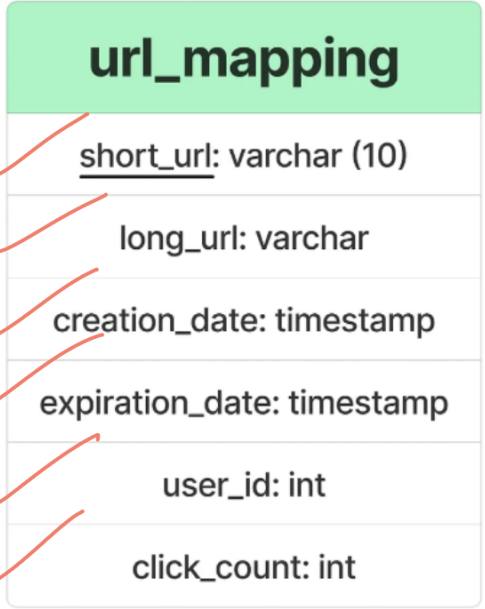
To choose the right database for our needs, let's consider some factors that can affect our choice:

- We need to store billions of records.
- Most database operations are simple key-value lookups.
- Read queries are much higher than write queries.
- We don't need joins between tables.
- Database needs to be highly scalable and available.

Given these points, a **NoSQL database** like **DynamoDB** or **Cassandra** is a better option due to their ability to efficiently handle billions of simple key-value lookups and provide high scalability and availability.

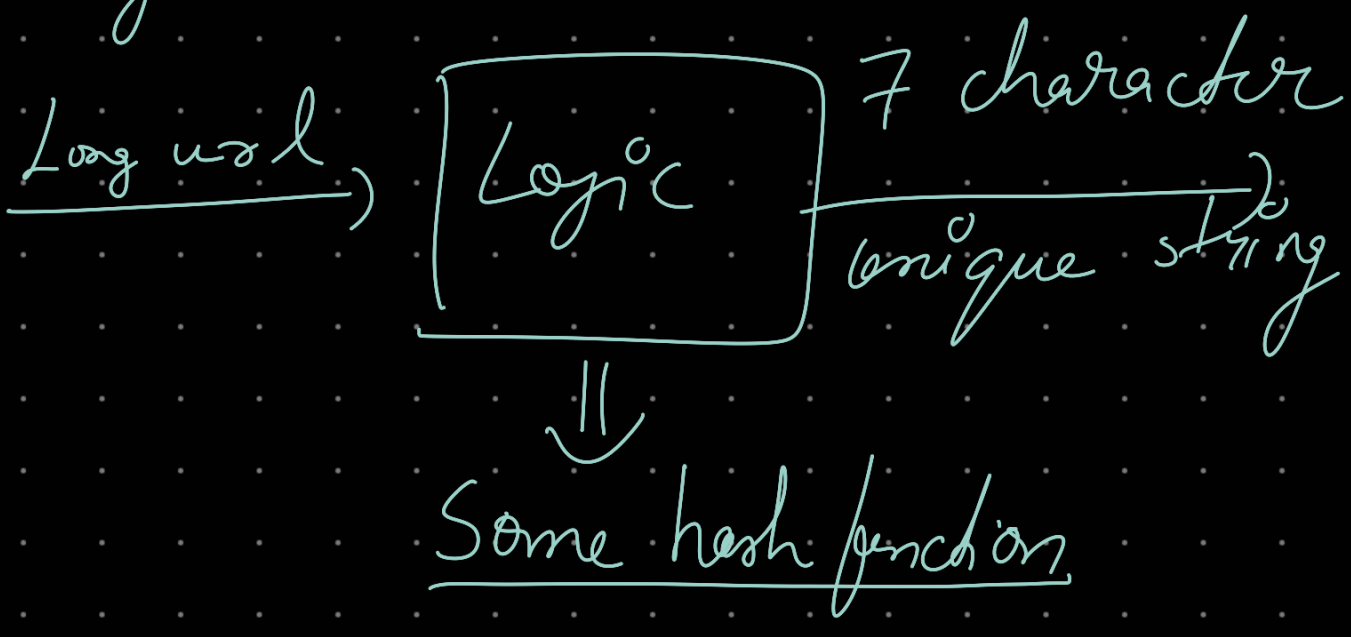
4.2 Database Schema

We would need two tables: one for storing url mappings and one for storing user related information.



(used if in case we need login flow as well)

* Now we have multiple servers, some problems may arise, lets see how url generation service logic.

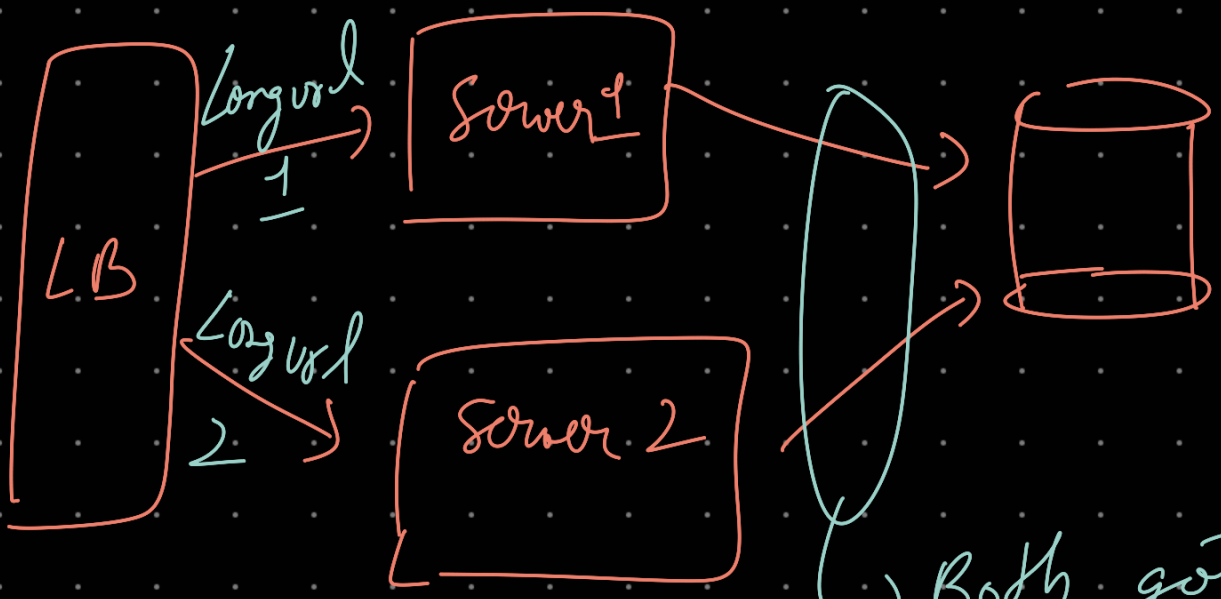


E.g → Long url → Hash → 6xm2bc1

Multiple long url can give same 7 characters

≠ characters

Collision



Both got same 7 charator hash

→ If we directly write to DB, we will have 2 different entries for same short url

⇓

On hitting short url → Long url 1 / Long url 2

Unpredictable redirect

* One solution, first check existence, then write
  - E.g. -> findOneAndUpdate with upsert true
  - Distributed locking

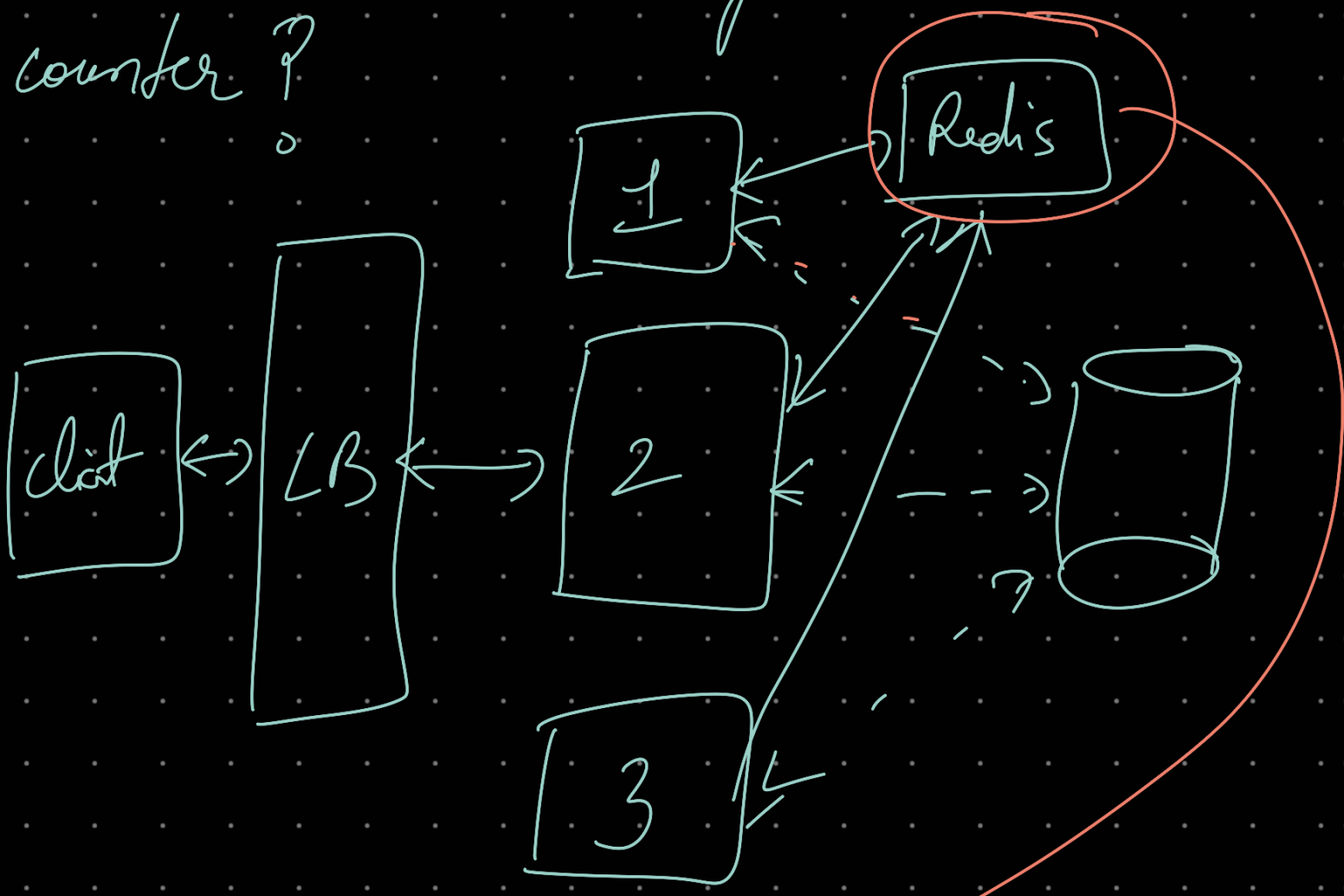
This works well even if we have more than 1 server instances or Machine

⇓

But this increases latency

⇓

Can we use Redis to generate a counter?



This redis will start from 0, 1, ... Billion, Trillion

↓

gives unique Number

↓

Use this number for hashing

Problem → 1 Redis → SPOF

↳ Adding Multiple redis

Redis 1 Redis 2

But both redis can give duplicate numbers → again some problem

We need to make sure they do not give some number

Solution

Range → (0 – 1 Million) (1M – 2M)

⇓

Fine, but adding new redis would be challenging, as we need to know which range is given to which Redis

⇓

Can use Apache Zookeeper

⇓

This contain mapping

Range	Redis
(0 – 1M)	R1
(1M – 2M)	R2

How the url generator work?

① Hashing + Encoding

↳ MD5/SHA-256

↳ Base 62

||

Uses: (A-Z, a-z, 0-9)

So, 7 character Base 62

↓

62 X - - - - - - 62

combination

Total permutation = 3.5 billion unique short url string

$= (62)^7$

Example Workflow:

1. User submits a request to generate short url for the long url: `https://www.example.com/some/very/long/url/that/needs/to/be/shortened`
2. Generate an MD5 hash of the long URL. MD5 produces a 128-bit hash, typically a 32-character hexadecimal string: `1b3aabf5266b0f178f52e45f4bb430eb`
3. Instead of encoding the entire 128-bit hash, we typically use a portion of the hash (e.g., the first few bytes) to create a more manageable short URL.
    a. First 6 bytes of the hash: `1b3aabf5266b`
4. Convert these bytes to decimal: `1b3aabf5266b` (hexadecimal) → `47770830013755` (decimal)
5. Encode the result into a Base62 encoded string: `DZFbb43`

The specific choice of 6 bytes (48 bits) is important because it produces a decimal number that typically converts to a Base62 string of approximately 7 characters.

↳ This can lead to collision which can be solved by counter approach

Link Expiration

Link expiration allows URLs to be valid only for a specified period, after which they become inactive.

Expiration Date Handling:

- **User-Specified Expiration**: Users can specify an expiration date when creating the short URL. The service should validate this date to ensure it's in the future and within allowable limits (e.g., not exceeding a maximum expiration period).
- **Default Expiration**: If no expiration date is provided, the service can assign a default expiration period (e.g., 1 year) or keep the link active indefinitely.

Expiration Logic:

- **Background Jobs**: A background job or cron job can be scheduled to periodically check for expired URLs and mark them as inactive or delete them from the database.
- **Real-Time Expiration**: During the redirection process, the service checks whether the URL has expired. If expired, the service can return an error message or redirect the user to a default page.

6.2 Redirection Service

When a user accesses a short URL, this service is responsible for redirecting the user to the original URL.

This involves two key steps:

- **Database Lookup**: The Service Layer queries the database to retrieve the original URL associated with the short URL. This lookup needs to be optimized for speed, as it directly impacts user experience.
- **Redirection**: Once the long URL is retrieved, the service issues an HTTP redirect response, sending the user to the original URL.

Example Workflow:

1. A user clicks on `https://short.ly/abc123`.
2. The Redirection Service receives the request and extracts the short URL identifier (`abc123`).
3. The service looks up `abc123` in the database or cache to find the associated long URL.
4. The service issues a 301 or 302 HTTP redirect response with the `Location` header set to the long URL (e.g., `https://www.example.com/long-url`).
5. The user's browser follows the redirect and lands on the original URL.

↳ store frequently accessed url in Redis

[API]

Endpoint: `POST /api/v1/shorten`

This endpoint creates a new short URL for a given long URL.

Sample Request:

```
{
  "long_url": "https://example.com/very/long/url/that/needs/shortening",
  "custom_alias": "",
  "expiry_date": "2024-12-31T23:59:59Z", // optional
  "user_id": "user123"
}
```

Sample Response:

```
{
  "short_url": "http://short.url/abC123",
  "long_url": "https://example.com/very/long/url/that/needs/shortening",
  "expiry_date": "2024-12-31T23:59:59Z",
  "created_at": "2024-08-10T10:30:00Z"
}
```

5.2 URL Redirection API

Endpoint: `GET /{short_url_key}`

↳ 7 character string

This endpoint redirects the user to the original long URL.

Sample Response:

```
HTTP/1.1 301 Moved Permanently
Location: https://www.example.com/some/very/long/url
```

↳ response header

Headers | Preview | Response | Initiator | Timing | Adblock | Cookies

▼ General

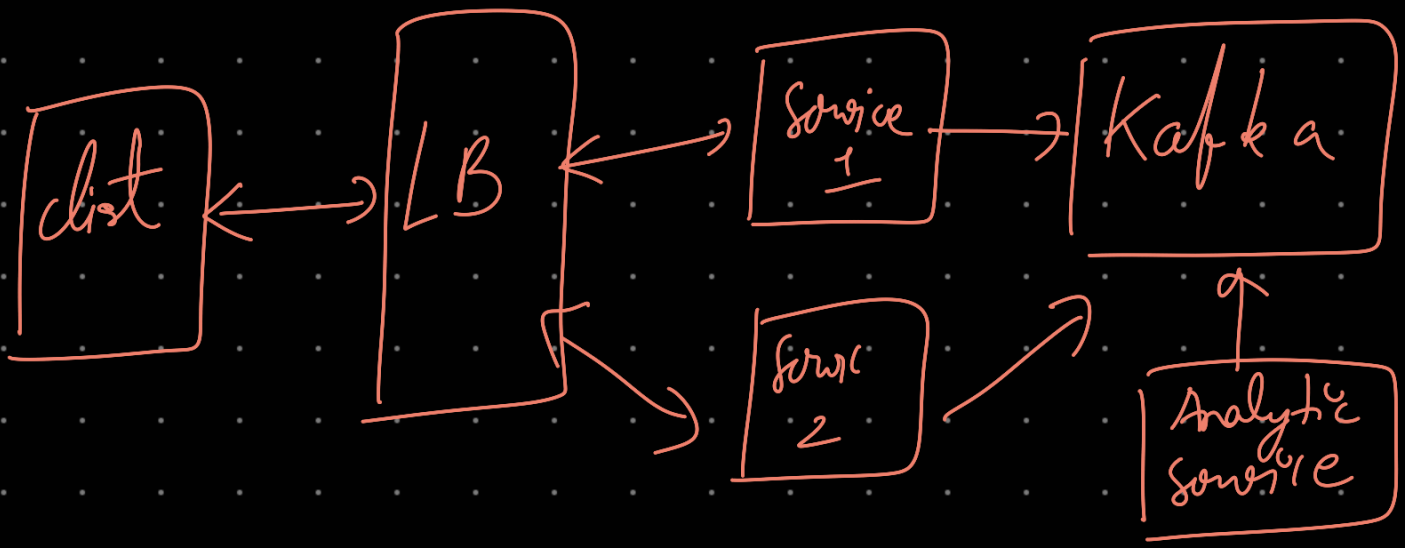
Request URL:	https://tinyurl.com/6amxhrb
Request Method:	GET
Status Code:	301 Moved Permanently
Remote Address:	104.18.111.161:443
Referrer Policy:	strict-origin-when-cross-origin

▼ Response Headers

Alt-Svc:	h3=":443"; ma=86400
Cache-Control:	max-age=0, must-revalidate, no-cache, no-store, private
Cf-Cache-Status:	EXPIRED
Cf-Ray:	8ee6cede1afcc188-BLR
Content-Type:	text/html; charset=utf-8
Date:	Sat, 07 Dec 2024 19:09:06 GMT
Location:	https://www.youtube.com/watch?v=fMZMm_0ZhK4
Referrer-Policy:	unsafe-url
Server:	cloudflare
Server-Timing:	cfCacheStatus;desc="EXPIRED"
Strict-Transport-Security:	max-age=31536000; includeSubDomains; preload
Vary:	Accept-Encoding
X-Content-Type-Options:	nosniff
X-Robots-Tag:	noindex
X-Tinyurl-Redirect:	eyJpdiI6IkkwQmtTN0Ird0xwNEdkS3JJdW1KcGc9PSIsInZhbHVlIjoic2JaMTNVTE JvOTZDa254WGFhVk9xandmekxiSDJyQmN3RDk4NnhjNHc9PSIsIm1hYyI6ImZ DYxYzExNTNhNTcxN2VlYjA1MTMiLCJ0YWciOiIifQ==
X-Tinyurl-Redirect-Type:	redirect
X-Xss-Protection:	1; mode=block

long url

How to design Analytics



6.3 Analytics Service

If the service needs to track analytics, such as the number of times a short URL is clicked, a separate analytics service can be introduced:

- Event Logging: Use a **message queue** (e.g., **Kafka**) to log each click event. This decouples the analytics from the core redirection service, ensuring that it doesn't introduce latency.
- Batch Processing: Process logs in batches for aggregation and storage in a **data warehouse** for later analysis.

7. Addressing Key Issues and Bottlenecks

7.1 Scalability

API Layer

Deploy the API layer across multiple instances behind a load balancer to distribute incoming requests evenly.

Sharding

Implement sharding to distribute data across multiple database nodes.

- **Range-Based Sharding:** If you are using an auto-incrementing ID as the shard key, the first shard might store IDs 1 to 1,000,000, the second shard 1,000,001 to 2,000,000, and so on.
  - **Limitations:** If your data isn't evenly distributed, one shard may become much larger than others, leading to uneven load distribution (known as a "hotspot").
- **Hash-Based Sharding:** It involves applying a hash function to the shard key to determine which shard the data should go to. For example, you might hash the short URL identifier and then take the modulo with the number of shards to determine the shard (e.g., `hash(short_url) % N` where `N` is the number of shards).
  - **Limitations:** When scaling out (adding new shards), re-hashing and redistributing data can be challenging and requires **consistent hashing** techniques to minimize data movement when adding or removing shards.

7.2 Availability

Replication

Use database replication to ensure that data is available even if some nodes fail.

Failover

Implement automated failover mechanisms for the API and data store layers to switch to backup servers in case of failure.

Geo-Distributed Deployment

Deploy the service across multiple geographical regions to reduce latency and improve availability.

7.4 Security

Rate Limiting

To prevent abuse (e.g., spamming the service with thousands of URLs), implement rate limiting at the API layer.

Input Validation

Ensure that the URLs being shortened do not contain malicious content.

HTTPS

All communication between clients and the service should be encrypted using HTTPS to prevent eavesdropping and man-in-the-middle attacks.

Monitoring and Alerts

Set up monitoring for unusual activity patterns and trigger alerts for potential DDoS attacks or misuse.