System Design – URL shortening service

Nitin Mishra

Step 1: Requirement Gathering

Functional requirements:

Generate short url from Long url

Redirect short url to long url

Performance requirements:

High Availability

Minimum Latency

Short urls to be unpredictable

Extended Requirements:

Analytics

Apis availability to enable third parties (B2B) integration

Step 2: Back of the envelope estmation

Assumption for scalability: 500M short urls being generated per month

Read write ratio assumption = 100:1

Expected No. of redirections = 100 * 500M = 50 Billion

URL redirections (Writes per second) = 500M / (30*24*3600) **= 200 URLs per second**

New URLs (Read per second) = 100 * 200 **= 20K URLs per second**

No. of max records (assuming 5 years expiry) = 500 Million * 5 years * 12 months = 30 Billion

Total storage required for 5 years (half KB per record)= 30 Billion * 500 bytes **= 15 TB**

Incoming Data Bandwidth (required for write) = 200 URLs * 500 bytes **= 100 KB/sec**

Outgoing Data Bandwidth (required for reads) = 20K URLS * 500 byte**s = 10MB/sec**

Expected Requests per day = 20K * 3600 sec * 24 Hours = 1.7 Billion

Cache Memory Required for 20% Hot URLs (80:20 RULE: assuming 20% of URLs bring 80% of the traffic) = 1.7 Billion * 20/100 * 500 bytes **= 170 GB**

Step 3: System APIs

createURL(api_dev_key,original_url,custom_alias=None,user_name=None,expire_date=None)(String,ERR)

deleteURL(api_dev_key, short_url_key)(SUCCESS, ERR)

Parameters:

api_dev_key (string): The API developer key of a registered account. This will be used to, among other things, throttle users based on their allocated quota.

original_url (string): Original URL to be shortened.

custom_alias (string): Optional custom key for the URL.

user_name (string): Optional user name to be used in the encoding.

expire_date (string): Optional expiration date for the shortened URL.

short_url_key (string): Shortened URL to be retrieved.

Prevent API Abuse: Api_dev_key will help in throttling no. of users per some time based on quota per api dev key

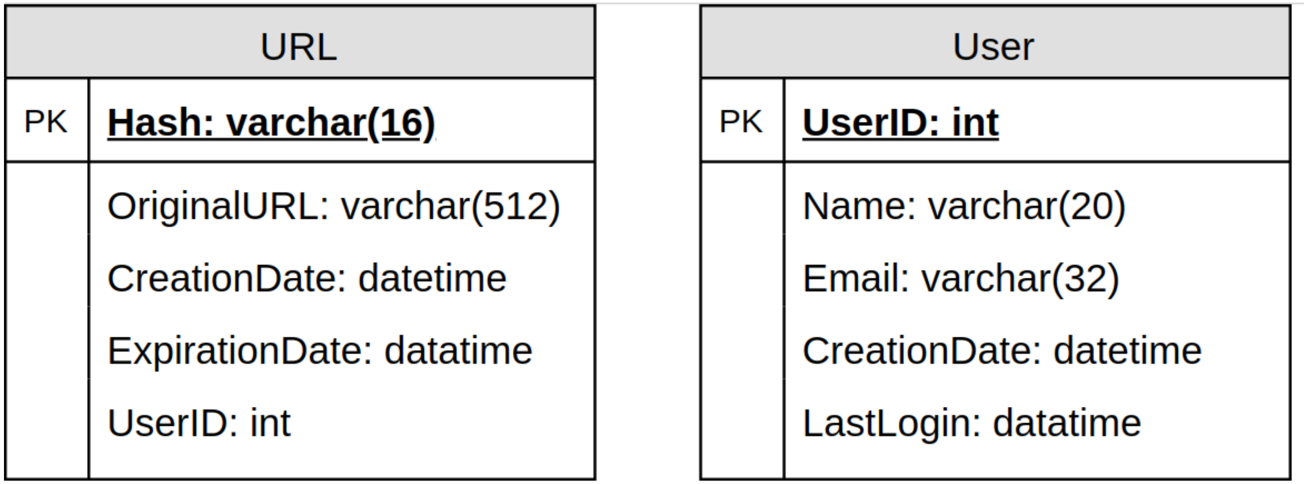
Step 4: Database Schema

We need to store: Billion of records

Nature of Data: Read-Heavy

Relationship required between records: No

Storage per obeject: less than 1KB



	URL
PK	**Hash: varchar(16)**
	OriginalURL: varchar(512) CreationDate: datetime ExpirationDate: datatime UserID: int

	User
PK	**UserID: int**
	Name: varchar(20) Email: varchar(32) CreationDate: datetime LastLogin: datatime

SQL or NoSQL?: Billions of rows and no joins required, NoSQL would be better choice.
Example: DynamoDB

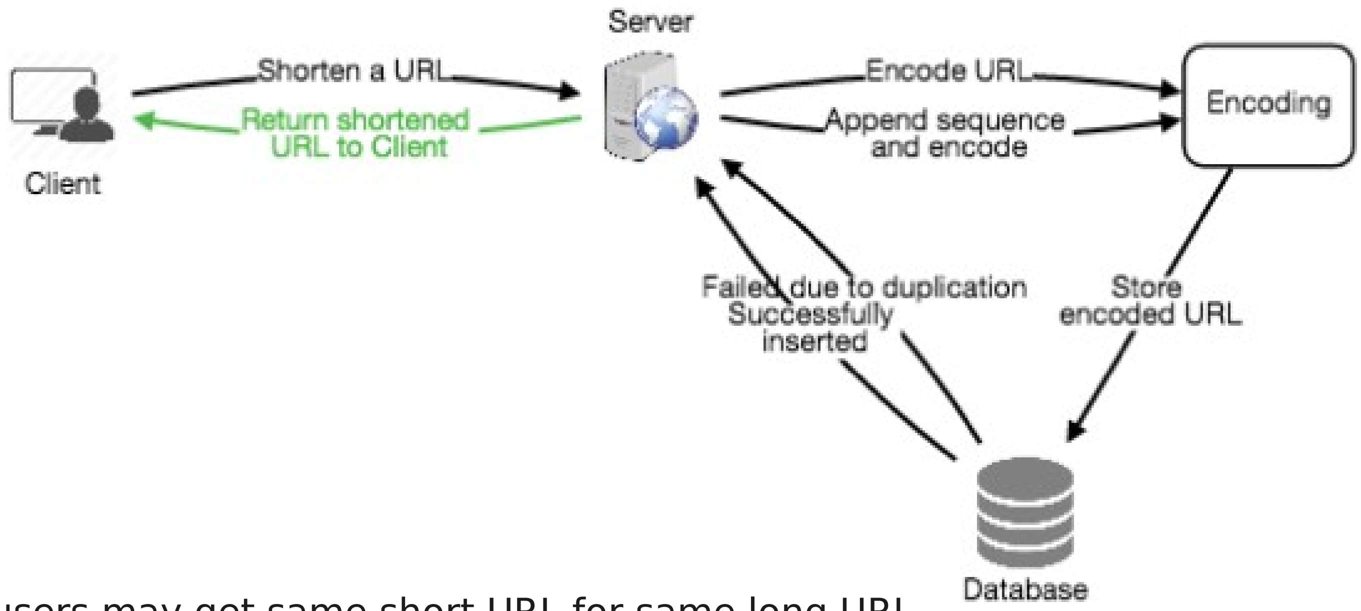
Step 5: URL shortening approach - 1st Approach for Key Generation (Encoding Actual URL)

Given Long URL -> MD5 Hashing (128 bit) -> Base64 Encoding ([A-Z, a-z, 0-9, +, /]) -> 21 char

Using base64 encoding, 6 letters long key would generate 64^6 = ~**68.7 billion strings**

Using base64 encoding, 8 letters long key would generate 64^8 = ~281 trillion strings

Conclusion: With 68.7B unique strings, let's assume six letter keys would suffice for our system.



Issue 1: Multiple users may get same short URL for same long URL.

Issue 2: Taking first 6 char out of 21 char would result in Key duplication

Request flow for shortening of a URL

Step 5: URL shortening approach - 2nd Approach for key generation (Offline Key Generation)

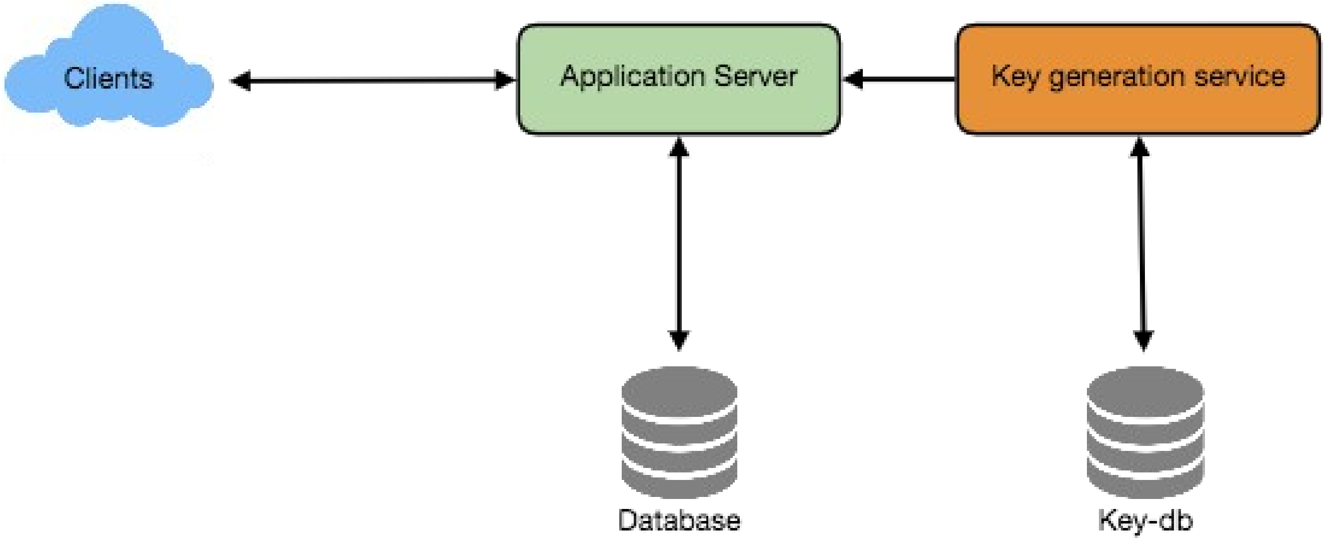
Key Generator Service (KGS): Storing unique random 6 letter key in advance.

Size of Key-DB = (6 characters per key * 68.7B unique keys **= 412 GB)**

Pros: No Need for encoding and no need to worry for duplication issue.

Cons: Single point of failure, Concurrency issue, Need to maintain 2 table (used and unused keys), More Storage required

Workaround: Need of standby replica of KGS (passive), Synchronization/ locking and caching



Step 6: Accessing Short URL Approach

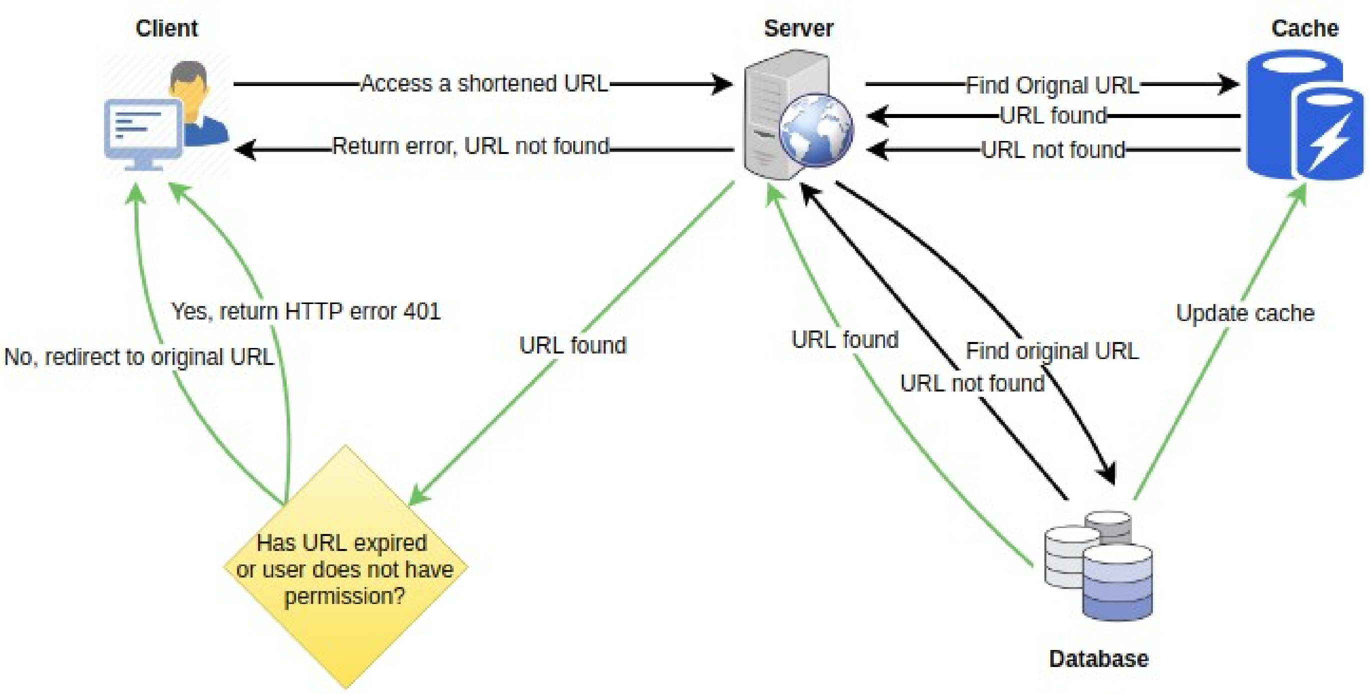
DB Partitioning Approaches:

1. Range Based Partitioning: Storing URLs on DB server having index as first letter of the key, but it leads to unbalanced server problem.

2. Hash Based Partitioning: Storing URLs on DB server having index as first letter of the hash of the key, here overloading can be solved using consistent hashing method.

Caching Frequently used URLs: Storing <hash, long url> in cache for 20% of daily traffic (around 170 GB) in single 256GB memory machine or replicate

Cache Eviction Policy: Least Recently used URL to be removed first would suffice.



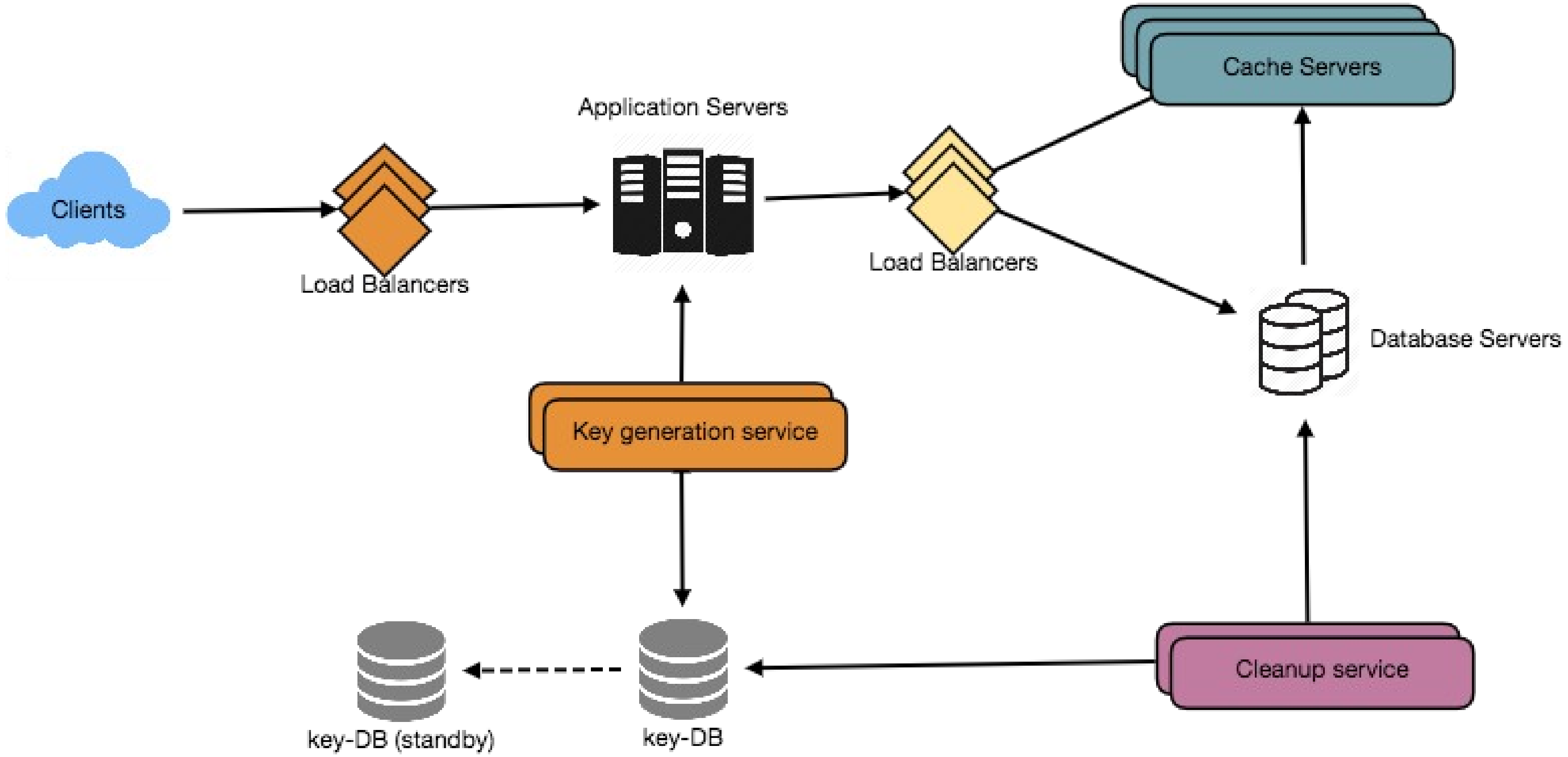
Request flow for accessing a shortened URL

Step 7: Final System Architecture

Making System More Reliable and Scalable:

1. Adding Load Balancers: Use LBs between Client, Application server, cache and DB with Weighted Round robin approach.

2. DB purge for old data: Separate cleanup service to delete expired URLs and put the key back in Key-DB.



Step 8: Analytics and Security

Analytics:

How many times a short URL has been used?
What were user locations, etc.?

Suggestion: Some statistics worth tracking: country of the visitor, date and time of access, web page that refers the click, browser, or platform from where the page was accessed.

Security and Permissions:

Can users create private URLs or allow a particular set of users to access a URL?

Suggestion: We can store the permission level (public/private) with each URL in the database. We can also create a separate table to store UserIDs that have permission to see a specific URL. If a user does not have permission and tries to access a URL, we can send an error (HTTP 401) back. Given that we are storing our data in a NoSQL wide-column database like Cassandra, the key for the table storing permissions would be the KGS generated ‘key’. The columns will store the UserIDs of those users that have the permission to see the URL.