International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320-088X

IMPACT FACTOR: 7.056

IJCSMC, Vol. 10, Issue. 5, May 2021, pg.95 – 101

Optimised URL Shortener using Cloud Platform

Vipin Wani¹; Aniket Paul²; Kaushik Kumar³

School of Computer Science & Engineering, Sandip University, Nashik, Maharashtra, India

vipin.wani@sandipuniversity.edu.in¹; aniketpaul446@gmail.com²; kr.kaushik1806@gmail.com³

DOI: 10.47760/ijcsmc.2021.v10i05.010

Abstract— Short URLs have become commonplace. Short URLs, which are especially popular in social networking services, have witnessed a significant increase in their use in recent years, largely due to Twitter's restriction on message length to 140 characters. We provide a first characterization in the usage of short URLs in this paper. Our specific goal is to create a system architecture for short URLs and have an optimized solution of how to use them and as well as their potential impact on web performance.

Our project is made while keeping in mind that the generated URL has to be one third the size of a given url. This allows the user the functionality to share the URL with multiple individuals. Our system also incorporates the idea of tracking each individual shortened URL by the usage of a short code. It gives the user the control over their generated URLs.

Keywords— Algorithm, URL, Cloud Computing, Load Balancer, Database.

I. INTRODUCTION

URL Shortening Has Become One of The Most Popular Methods for Exchanging and Disseminating URL. URL Shortening Services Give Its Consumers a Smaller Version of Any Long URL They Provide, And Then Redirect Visitors to The Intended Source. Although The First Prominent URL Shortening Service, Tiny URL, Was Launched in 2002, Today's Users Have a Wide Range of Options. Shortening Services Have Recently Gained Popularity as a Result of Their Widespread Use in Online Social Networks (Sons).

Because Some Services, Such as Twitter, Limit the Length of Messages That Can Be Posted, URL Shortening Is Common for Content Distribution. While Short URL Accesses Account for A Small Percentage of a Site's "Web Hits," They Are Fast Increasing by As Much as 10% [1].

We Believe That Understanding How They Are Used Is Critical for A Variety of Reasons.

1. Short URLs Are Popular in Niche Communities and Services Like Twitter, as Well as in A Variety of Online Social Networks and Instant Messaging (IM) Systems. A Study of URL Shortening Services Will Provide Insight into Such Communities' Interests as Well as A Better Knowledge of Their Characteristics in Comparison to The Broader Online Browsing Population [2].

2. Some URL Shortening Services, Such as Bit. Ly, Have Become So Popular That They Now Account for As Much as 1% of the Total Web Population on A Daily Basis [3]. If This Trend Continues, URL Shortening Services Will Become a Vital Component of The Web's Infrastructure, Raising Concerns About Its Performance, Scalability, And Dependability.
3. We Feel It Is Impossible to Answer Issues About Usage and Design Effective Designs for URL Shortening Services Without First Analyzing Their Access Patterns.

II. UNDERSTANDING THE REQUIREMENTS

To Make Sure Our System and Design Works as Expected We Have Outlined a Couple of Requirements That Will Benefit the User.

1. Our Service Should Be Able to Generate a Shorter and More Distinctive Alias for Every Given URL.
2. Our Service Should Redirect Customers to The Original Link When They Click on A Short Link.
3. Users Should Be Able to Choose a Custom Short Link for Their URL as An Option.
4. Links Will Expire After a Set Amount of Time.
5. The Expiration Time Should Be Configurable by The User.
6. The System Should Have a High Level of Availability. This Is Necessary Because If Our Service Is Unavailable, All URL Redirections Would Fail.
7. URL Redirection Should Take Place in Real Time with The Least Amount of Latency Possible.

III. ESTIMATION OF CAPACITY

Ours System Will Be a Reading-Intensive System. In Comparison to New URL Shortenings, There Will Be Many Redirection Requests. Assume That the Read and Write Ratios Are 100:1.

A. Traffic Estimates

If We Assume 100 million New URL Shortenings Per Month and A 100:1 Read/Write Ratio, We Can Expect 10 billion Redirections in The Same Time Period:

$$100 * 100M = 10B$$

Number Of Queries Our System Has to Handle to Keep Up with It Will Be

$$100 \text{ million} / (30 \text{ days} * 24 \text{ hours} * 3600 \text{ seconds}) = \sim 38 \text{ URLs/s}$$

With A Read/Write Ratio Of 100:1, The Number of URL Redirections Per Second Will Be:

$$100 * 38 = 138/s$$

Let's Say We Keep Track of Every URL Shortening Request (And the Abbreviated Link That Goes with It) For Five Years. With 100 million New URLs Expected Per Month, The Total Number of Items We Intend to Store Will Be 6 billion.

$$100 \text{ million} * 5 \text{ years} * 12 \text{ months} = 6 \text{ billion}$$

Let's Assume That Each Stored Object Is 500 Bytes in Size (This Is Just a Guess--We'll Look at It Later). We'll Need a Total Of 3TB Of Storage:

$$6 \text{ billion} * 500 \text{ bytes} = 3 \text{ TB}$$

B. Bandwidth Estimates

Because We Predict 38 New URLs Every Second for Write Requests, The Total Incoming Data for Our Service Will Be 100 KB Per Second

$$38 * 500 \text{ bytes} = 19 \text{ KB/s}$$

Because We Estimate 138 URL Redirections Every Second for Read Requests, The Total Outgoing Data for Our Service Would Be 0.69MB Per Second:

$$138 * 500 \text{ bytes} = \sim 0.69 \text{ MB/s}$$

IV. INTRODUCTION OF ENCODING IN THE SYSTEM

When Employed as A Hash Function, The MD5 Algorithm Generates A 128-Bit Hash Result. After Base64 Encoding, (e.g., Fig. 1) We'll Have a String Containing More Than 21 Characters (Since Each Base64 Character Encodes 6 Bits of The Hash Value). However, This May Result in Key Duplication; To Avoid This, We Can Swap Out Specific Characters or Choose Characters from The Encoding String [4]. This Problem Will Lead to Many Users Entering the Same URL and The Abbreviated URL Will Be the Same, Which Is Against the Requirement We Highlighted.

To Make Each Input URL Unique, We Can Append an Increasing Sequence Number to It and Then Compute Its Hash. However, We Don't Need to Save This Sequence Number in The Databases. An Ever-Increasing Sequence Number Could Be a Concern with This Method.

Another Option Is to Append the User Id to The Input URL (Which Should Be Unique). If The User Hasn't Signed in Yet, We'll Have to Prompt Them to Select a Uniqueness Key. If There Is Still a Disagreement, We Must Keep Creating Keys Until We Find One That Is Unique.

Fig 1.0 Encoding Diagram

A Standalone Key Generation Service (KGS) [5] That Creates Random Six-Letter Strings and Saves Them in A Database (Let's Name It Key-DB) Is Possible. We'll Utilize One of The Already-Generated Keys to Abbreviate A URL Whenever We Need To.

This Method Simplifies and Expedites the Process. We Won't Have to Worry About Duplications or Collisions Because the URL Won't Be Encoded. KGS Will Ensure That All Keys Placed into Key-DB Are One-Of-A-Kind [5].

Once A Key Has Been Used, It Should Be Recorded in The Database to Prevent It from Being Used Again. If Many Servers Are Reading Keys at The Same Time [6], We May See a Situation Where Two Or More Servers Attempt to Read the Same Key from The Database.

KGS Can Be Used by Servers to Read and Mark Keys in Databases. KGS Can Keep Keys in Two Tables: One for Keys That Haven't Been Used Yet, And Another for All Keys That Have Been Used. KGS Can Move Keys to The Used Keys Table as Soon as They Are Given to One of The Servers. KGS Can Store a Set of Keys in Memory at All Times in Case A Server Need Them.

For Simplicity, KGS Can Move Keys to The Used Keys Table as Soon as They Are Loaded into Memory. This Guarantees That Each Server Has Its Own Set of Keys. We Will Be Squandering Those Keys If KGS Dies Before Allocating All of The Loaded Keys to Some Server–Which May Be Acceptable Given the Large Number of Keys We Have.

KGS Must Also Ensure That the Same Key Is Not Used by Several Servers. Before Removing Keys from The Data Structure and Delivering Them to A Server, It Must Synchronize (Or Gain a Lock On) The Data Structure Holding the Keys.

We Can Construct 68.7 billion Unique Six-Letter Keys Using Base64 Encoding. We Can Store All of These Keys in One Byte If We Use One Byte to Hold One Alpha-Numeric Character In:

$$6 \text{ (characters per key)} * 68.7B \text{ (unique keys)} = 412 \text{ GB}$$

Custom aliases are supported by our system. Users can pick any 'key' they like, but providing a custom alias is not mandatory. However, it is reasonable (and often desirable) to impose a size limit on a custom alias to ensure we have a consistent URL database.

V. PARTITIONING AND REPLICATION OF DATA

We Need to Split Our Database Such That It Can Hold Information About Billions of URLs In Order to Scale It Out. As A Result, We'll Need to Devise a Partitioning Strategy That Divides and Stores Our Data Over Multiple DB Servers.

Ranged Based Partition: We May Store URLs In Several Partitions Based on The Initial Letter of The Hash Key. As A Result, All URLs That Begin with The Letter 'A' (And 'A') Are Recorded in One Partition, Followed by Those That Begin with The Letter 'B', And So On. Range-Based Partitioning Is the Name for This Method. We Can Even Put A Few Unusual Characters Together in One Database Area. As A Result, We Need Create a Static Partitioning System for Reliably Storing and Retrieving URLs [7].

The Most Serious Disadvantage of This Strategy Is That It Might Lead to Imbalanced Database Servers. Let's Say We Decide to Put All URLs That Start with The Letter "E" Into A Database Partition, Only to Find Out Later That We Have Far Too Many "E" URLs.

Hash Based Partitioning: We Take a Hash of The Object We're Storing in This Scheme. The Hash Is Then Used to Determine Which Partition to Use. In Our Situation, The Hash of the 'Key' Or the Short Link Can Be Used to Determine the Partition in Which the Data Object is Stored [8].

Our Hashing Function Will Distribute URLs Into Multiple Partitions at Random (For Example, any 'Key' Can Be Mapped to A Number Between [1...256]). This Number Represents the Division in Which Our Object Is Stored.

VI. USAGE OF CACHING MECHANISM

URLs That Are Often Visited Can Be Cached. We May Use Any Off-The-Shelf Solution, Such as Memcached, To Store Entire URLs Along with Their Hashes. As A Result, Before Hitting the Backend Storage, The Application Servers Can Quickly Check If the Cache Has the Desired URL.

To Start with Cache Memory Requirements, We May Start with 20% Of Daily Traffic and Change the Number of Cache Servers as Needed Based on Client Usage Patterns. To Cache 20% Of Daily Traffic, We'll Require 170GB of RAM, As Previously Calculated. We Can Easily Fit All of The Cache into One Machine Because a Modern-Day Server Can Have 256GB of RAM. Alternatively, We May Store All of These Popular URLs On A Few Of Smaller Servers.

To Deal with When the Cache Is Full and We Need to Change a Link with A Newer/Hotter URL. For Our System, LRU (Least Recently Used) Can Be a Suitable Policy. We Start with The URL That Has Been Used the Least Lately. To Store Our URLs And Hashes, We May Use a Linked Hash Map or A Similar Data Structure, Which Will Also Keep Track of The URLs That Have Been Viewed Recently as shown in figure 2.0.

We May Replicate Our Cache Servers to Divide the Load Amongst Them to Boost Efficiency Even Further.

Our Servers Would Hit a Backend Database Whenever a Cache Miss Occurred. We May Update the Cache and Transmit the New Entry to All Cache Replicas Once This Happens. By Adding The New Entry, Each Copy May Update Its Cache. If The Entry Already Exists in A Replica, It Can Be Ignored [9].

Fig 2.0 URL lifecycle.

VII. USAGE OF LOAD BALANCER

We May Add a Load Balancing Layer to Our System in Three Places:

1. Between Clients and Application Servers
2. Between Application Servers and Database Servers
3. Between Application Servers and Cache Servers

We Start with A Basic Round Robin Strategy [10], Which Evenly Distributes Incoming Requests Among Backend Servers. This Load Balancing Is Easy to Set Up and Doesn't Add Any More Overhead. Another Advantage of This Method Is That If a Server Goes Down, Load Balancing Removes It from The Rotation and Stops Transmitting Traffic to It. One Issue with Round Robin LB Is That We Don't Account for Server Load. As A Consequence, Even If a Server Is Overburdened or Sluggish, The LB Will Continue to Send New Requests to It. To Deal with This, A More Intelligent LB Solution May Be Implemented, Which Polls the Backend Server About Its Load On a Regular Basis and Adjusts Traffic Accordingly as shown in figure 3.0

VIII. DATABASE CLEAN-UP OF EXPIRED LINKS

Searching For Expired Links on A Regular Basis Would Place a Lot of Strain on Our Database. Instead, We May Execute a Lazy Cleaning and Gently Delete Expired Links. Only Expired Links Will Be Erased by Our Service, While Some Expired Links May Exist Longer but Will Never Be Restored to Users.

Steps That Can Be Taken:

1. We Can Erase an Expired Link and Provide an Error to A User Whenever They Try to Access It.
2. Cleanup Is a Separate Service That May Run on A Regular Basis to Remove Expired Links from Our Store and Cache. This Service Should Be Very Light, And It Should Only Be Performed When User Traffic Is Predicted to Be Light.
3. Each Link Can Have a Preset Expiration Time (E.G., Two Years).
4. We May Put the Key Back in The Key-DB to Be Utilized After Deleting an Expired Link.

Fig 3.0 Integration of Load balancers.

IX. PERMISSIONS AND SECURITY

In The Database, We May Store the Permission Level (Public/Private) For Each URL. We Can Also Construct a Separate Table to Keep Track of Use rids With Access to A Given URL. We Can Return an Error (HTTP 401) If A User Does Not Have Authorization and Attempts to Visit A URL.

Given That We're Using a NoSQL Wide-Column Database Like Cassandra to Store Our Data, the 'Hash' (Or the KGS Produced 'Key') Would Be the Key for The Table Containing Permissions. The Use rids Of Those Users Who Have Authorization to See the URL Will Be Stored in The Columns.

X. CONCLUSION

We have presented a large-scale study of URL shortening services by exploring traces both from the services themselves and from one of the largest pools of short URLs, social network. We have gone over multiple designs, approaches and came up with set of configurations that takes care of the Encoding, Database Cleanup, Caching and Security.

REFERENCES

- [1]. March 2011 Conference: Proceedings of the 20th International Conference on *World Wide Web*, *WWW 2011*, Hyderabad, India, March 28 - April 1, 2011 DOI:10.1145/1963405.1963505 [Online] Available: <https://doi.org/10.1145/1963405.1963505>
- [2]. March 2011 *we.b: the web of short URLs*, *ACM The Digital library* [Online] Available: <https://dl.acm.org/doi/10.1145/1963405.1963505>
- [3]. March 2011 *we.b*, Research Gate [Online] Available: <https://www.researchgate.net/publication/221023769>
- [4]. January 2020 Designing a URL Shortening service like Tiny URL [Online] Available: <https://stackoverflow.com/questions/59692297>
- [5]. July 2019 What is KGS? Why 2 table? [Online] Available: <https://discuss.educative.io/t/what-is-kgs-why-2-table/12665>
- [6]. 2017 Bad Practices in Database Design: [Online] Available: <https://www.toptal.com/database/database-design-bad-practices>
- [7]. Designing a URL Shortening service: educative [Online] Available: <https://www.educative.io/courses/grokking-the-system-design-interview/m2ygV4E81AR>
- [8]. *Database VLDB and Partitioning Guide* [Online] Available: <https://docs.oracle.com/database/121/VLDBG/GUID-F023D3ED-262F-4B19-950A-D3C8F8CDB4F4.htm#VLDBG1270>
- [9]. *What Is a Cache Miss?* [Online] Available: <https://hazelcast.com/glossary/cache-miss>
- [10]. 2014 *Round Robin Scheduling* [Online] Available: <https://www.sciencedirect.com/topics/computer-science/round-robin-scheduling>